elektor books

Getting Started With Java Using Eclipse

Mastering the Language and the Development Platform

Bernhard Steppan

elektor
design > share > earn

[Bernhard Steppan]

[Getting Started With Java Using Eclipse]

[Mastering the Language and the Development Platform]

[Bernhard Steppan]

[Getting Started With Java Using Eclipse]

[Mastering the Language and the Development Platform]

This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11, NL-6144-ZG, Susteren, The Netherlands.
Telephone: +31 46 4389444

All information, procedures and illustrations contained in this book have been prepared to the best of our knowledge and tested with care. Nevertheless, errors cannot be completely ruled out. For this reason, the information contained in this book is not associated with any obligation or guarantee of any kind. Consequently, the authors and publishers assume no responsibility and will not accept any liability, consequential or otherwise, arising in any way from the use of this program material - or any part thereof. Likewise, authors and publishers do not guarantee that described procedures etc. are free of third party intellectual property rights. The reproduction of common names, trade names, product designations, etc. in this work does not justify the assumption that such names are to be considered free in the sense of trademark and brand protection legislation and may therefore be used by anyone, even without special identification.

British Cataloguing in Publication Data
A catalogue record for this book is available at the British library.

Print: ISBN 978-3-89576-561-2
E-book: ISBN 978-3-89576-562-9

This work is protected by copyright.
All rights reserved, including those of translation, reprint and reproduction of the book or parts thereof. No part of this work may be reproduced in any form (photocopy, microfilm or any other process), not even for the purpose of teaching - with the exception of the special cases mentioned in §§ 53, 54 URG -, or processed, copied or distributed using electronic systems without the written permission of the publisher.

© Copyright 2023: Elektor International Media B.V., www.elektor.com
Prepress Production: Bernard Steppan
Editors: Alina Neacsu; Jan Buiting

Contents

Preface XXI

Part I Basics 1

1 Programming Basics 3
1.1 Introduction 3
1.2 The Language of the Machine World 4
1.3 High-Level Programming Languages 5
1.4 Development Environment 7
1.4.1 Compiler 7
1.4.2 Editor 7
1.4.3 Project Management 7
1.5 Runtime Environment 8
1.6 Summary 8
1.7 Literature 9
1.8 Exercises 9

2 Technology Overview 11
2.1 Introduction 11
2.2 Overview 12
2.2.1 The Early Days of Java 12
2.2.2 The Growth Period of Java 13
2.2.3 The Presence And Future of Java 14
2.3 Why Java? 15
2.3.1 Easy to Read 15
2.3.2 Object-Oriented 15
2.3.3 Safe And Robust 15

2.3.4 Very Powerful 16
2.3.5 Universally Useable 16
2.3.6 Free of Charge 16
2.3.7 Open Source 16
2.3.8 Easily Portable 16
2.3.9 Easily Expandable 17
2.3.10 Easy to Develop And Test 17
2.4 What Belongs to Java? 18
2.4.1 Java Programming Language 18
2.4.2 Java Virtual Machine 18
2.4.3 Java Class Libraries 20
2.4.4 Java Development Tools 20
2.5 Java Versions 21
2.6 Java Editions 21
2.6.1 Java Standard Edition 21
2.6.2 Java Enterprise Edition 21
2.6.3 Java Micro Edition 21
2.7 Summary 22
2.8 Literature 22
2.9 Exercises 22

3 Object-Oriented Programming 23
3.1 Introduction 23
3.2 Overview 24
3.3 Object 25
3.4 Class 26
3.4.1 Properties 26
3.4.2 Methods 28
3.5 Abstraction 30
3.6 Inheritance 31
3.6.1 Base Classes 32
3.6.2 Derived Classes 33
3.6.3 Multiple Inheritance 33
3.7 Access Protection 34
3.8 Relationships 36
3.8.1 Relationships Without Inheritance 36
3.8.2 Inheritance Relationships 38
3.9 Design Flaws 40

3.10 Refactoring 41
3.11 Modeling 41
3.12 Persistence 41
3.13 Polymorphism 41
3.13.1 Static Polymorphism 42
3.13.2 Dynamic Polymorphism 42
3.14 Design Rules 43
3.15 Summary 43
3.16 Literature 44
3.17 Exercises 44

4 Development Environment 45
4.1 Introduction 45
4.2 Installation 46
4.2.1 Operating System 46
4.2.2 Install Java 47
4.2.3 Install Eclipse 50
4.2.4 Install Sample Programs 55
4.2.5 Installation Check 58
4.3 Eclipse Introduction 60
4.3.1 Overview 60
4.3.2 Workbench 60
4.3.3 Perspectives, Views and Editors 61
4.3.4 Package Explorer 62
4.3.5 Java Editor 63
4.3.6 Code Formatter 66
4.3.7 Build System 68
4.3.8 Debugger 68
4.3.9 Modular Structure 69
4.3.10 Eclipse Workspace 70
4.3.11 Software Update 72
4.3.12 Help System 73
4.4 Summary 74
4.5 Literature 74
4.6 Exercises 75

Part II Java Language 79

5 Program Structure 81
5.1 Introduction 81
5.2 Overview 82
5.3 Language Elements 83
5.3.1 Comments 84
5.3.2 Packages 84
5.3.3 Classes 84
5.3.4 Methods 85
5.3.5 Statements 86
5.4 Structure of the Program 88
5.5 Program Flow 89
5.6 Reserved Keywords 90
5.7 Summary 91
5.8 Tutorial 92
5.9 Exercises 92

6 Variables 93
6.1 Introduction 93
6.2 Overview 94
6.2.1 Variable Purpose 94
6.2.2 Variable Types 94
6.2.3 Variable Usage 95
6.3 Local Variables 97
6.4 Parameters 98
6.5 Instance Variables 99
6.5.1 Individual Instance Variables 99
6.5.2 Instance Variable »this« 100
6.6 Class Variables 102
6.7 Constants 104
6.8 Summary 105
6.9 Literature 106
6.10 Tutorial 106
6.11 Exercises 107

7 Statements 109
7.1 Introduction 109
7.2 Overview 110
7.2.1 Statement Purpose 110
7.2.2 Statement Types 111
7.3 Declaration 111
7.4 Assignment 113
7.4.1 Java Assignment Structure 113
7.4.2 Java Assignments Are Not Equal to Mathematical Equations 113
7.4.3 Is x = y Equal to y = x? 114
7.4.4 Combination of Declaration and Value Assignment 115
7.5 Block 116
7.6 Variable Call 119
7.7 Method Call 120
7.8 Summary 121
7.9 Literature 122
7.10 Tutorial 122
7.11 Exercises 122

8 Primitive Data Types 123
8.1 Introduction 123
8.2 Overview 124
8.2.1 Purpose of Primitive Data Types 124
8.2.2 Types of Primitive Data Types 124
8.2.3 Use of Primitive Data Types 124
8.3 Integer Data Types 128
8.3.1 Data Type »byte« 128
8.3.2 Data Type »short« 129
8.3.3 Data Type »int« 130
8.3.4 Data Type »long« 131
8.4 Floating Point Data Types 131
8.4.1 Data Type »float« 132
8.4.2 Data Type »double« 133
8.5 Character Data Type 134
8.6 Boolean Data Type 134
8.7 Summary 135
8.8 Literature 136
8.9 Tutorial 136
8.10 Exercises 137

9 Classes and Objects 139
9.1 Introduction 139
9.2 Overview 140
9.2.1 Purpose of a Class 140
9.2.2 Types of Classes 141
9.2.3 Definition of Classes 141
9.2.4 Use of Classes 142
9.3 Anonymous Classes 144
9.3.1 Definition of Concrete Classes 144
9.3.2 Creation of Objects of a Concrete Class 146
9.3.3 Inner Classes 147
9.3.4 Local Classes 149
9.3.5 Anonymous Classes 150
9.3.6 Inheritance 152
9.4 Abstract Classes 157
9.5 Interfaces 158
9.6 Generics 162
9.6.1 Definition of Generic Classes 162
9.6.2 Creation of Generic Objects 163
9.7 Summary 167
9.8 Literature 167
9.9 Tutorial 167
9.10 Exercises 168

10 Enumerations 169
10.1 Introduction 169
10.2 Overview 170
10.2.1 Purpose of Enums 170
10.2.2 Definition And Declaration of Enums 171
10.2.3 Usage of Enums 172
10.3 Enum Classes 173
10.3.1 Constructor 174
10.3.2 Method »value()« 174
10.3.3 Separate Simple Enum Class 174
10.3.4 Separate Extended Enum Class 176
10.3.5 Inner Extended Enum Class 177
10.4 Summary 178
10.5 Literature 178
10.6 Tutorial 179
10.7 Exercises 179

11 Arrays 181
11.1 Introduction 181
11.2 Overview 182
11.2.1 Purpose of Arrays 182
11.2.2 Types of Arrays 182
11.2.3 Usage of Arrays 183
11.3 Summary 187
11.4 Tutorial 188
11.5 Exercises 188

12 Methods 189
12.1 Introduction 189
12.2 Overview 190
12.2.1 Method Purpose 190
12.2.2 Method Types 191
12.2.3 Method Definition 192
12.2.4 Method Usage 195
12.3 Constructors 197
12.3.1 Default Constructors 198
12.3.2 Constructors Without Parameters 199
12.3.3 Constructors With Parameters 200
12.4 Destructors 202
12.5 Operations 204
12.6 Getter Methods 205
12.6.1 Definition 205
12.6.2 Usage 207
12.7 Setter Methods 208
12.7.1 Definition 208
12.7.2 Usage 209
12.8 Summary 210
12.9 Literature 211
12.10 Tutorial 211
12.11 Exercises 211

13 Operators 213
13.1 Introduction 213
13.2 Overview 214
13.2.1 Operator Types 214
13.2.2 Operator Precedence 214

13.3 Arithmetic Operators 215
13.3.1 Unary Plus Operator 215
13.3.2 Unary Minus Operator 216
13.3.3 Addition Operator 217
13.3.4 Subtraction Operator 218
13.3.5 Multiplication Operator 219
13.3.6 Division Operator 219
13.3.7 Remainder Operator 220
13.3.8 Pre-increment Operator 220
13.4 Relational Operators 223
13.4.1 »Equal to« Operator 224
13.4.2 »Not Equal to« Operator 224
13.4.3 »Less Than« Operator 225
13.4.4 »Less Than or Equal to« Operator 225
13.4.5 »Greater Than« Operator 226
13.4.6 »Greater Than Or Equal To« Operator 226
13.4.7 Type Comparison Operator 227
13.5 Logical Operators 230
13.5.1 Logical Complement Operator 230
13.5.2 AND Operator 230
13.5.3 OR Operator 232
13.5.4 Ternary Operator 232
13.6 Bitwise Operators 233
13.7 Assignment Operators 233
13.8 New Operator 234
13.9 Cast Operator 235
13.10 Access Operators 237
13.10.1 Dot Operator 237
13.10.2 Lambda Operator 239
13.11 Summary 240
13.12 Literature 240
13.13 Tutorial 240
13.14 Exercises 240

14 Conditional Statements 241
14.1 Introduction 241
14.2 Overview 242
14.3 »If Then Else« Statement 242

14.4 Ternary Operator 243
14.5 Switch Statement 245
14.5.1 Switch Statement at the Level of Java 6 245
14.5.2 Switch Statement at the Level of Java 7 246
14.5.3 Yield Statement 247
14.5.4 Lambda Expression 248
14.6 Summary 249
14.7 Literature 249
14.8 Tutorial 249
14.9 Exercises 249

15 Loops 251
15.1 Introduction 251
15.2 Overview 252
15.2.1 Purpose of Loops 252
15.2.2 Types of Loops 252
15.3 While Loop 253
15.4 Do Loop 254
15.5 Simple For Loop 255
15.6 Extended For Loop 256
15.7 Summary 257
15.8 Literature 257
15.9 Tutorial 258
15.10 Exercises 258

16 Packages and Modules 259
16.1 Introduction 259
16.2 Overview 260
16.3 Packages 260
16.3.1 Class Import 260
16.4 Modules 264
16.5 Summary 267
16.6 Tutorial 267
16.7 Exercises 267

17 Exception Handling 269
17.1 Introduction 269
17.2 Overview 270
17.2.1 Motivation 270

17.2.2 Types of Errors ... 270
17.2.3 Use of Exception Handling ... 270
17.3 Base Class »Throwable« ... 273
17.4 Class »Error« ... 274
17.4.1 Subclass »OutOfMemoryError« ... 275
17.4.2 Subclass »StackOverflowError« ... 277
17.5 Class »Exception« ... 278
17.5.1 Subclass »RuntimeException« ... 278
17.5.2 Subclass »IOException« ... 278
17.5.3 Self-Programmed Exceptions ... 279
17.6 Summary ... 282
17.7 Literature ... 282
17.8 Tutorial ... 283
17.9 Exercises ... 283

18 Documentation ... 285
18.1 Introduction ... 285
18.2 Overview ... 286
18.3 Line Comments ... 286
18.4 Block Comments ... 287
18.5 Documentation Comments ... 287
18.6 Summary ... 288
18.7 Literature ... 289
18.8 Tutorial ... 289
18.9 Exercises ... 290

19 Annotations ... 291
19.1 Introduction ... 291
19.2 Overview ... 292
19.2.1 Annotation Purposes ... 292
19.2.2 Annotation Types ... 292
19.2.3 Predefined Annotations ... 293
19.2.4 Use of Annotations ... 294
19.3 Compiler Control Annotations ... 296
19.3.1 Annotations »Deprecated« ... 296
19.3.2 Annotations »SuppressWarnings« ... 298
19.3.3 Annotation »Override« ... 304
19.4 Summary ... 307
19.5 Literature ... 307
19.6 Tutorial ... 307
19.7 Exercises ... 307

Part III Java Technology 309

20 Development Processes 311
20.1 Introduction 311
20.2 Overview 312
20.2.1 Correlation Between Phases And Activities 313
20.2.2 Activities 314
20.2.3 Tools 315
20.3 Planning Phase 315
20.3.1 Order Clarification 315
20.3.2 Requirements Capture 315
20.4 Construction Phase 316
20.4.1 Analysis 316
20.4.2 Design 317
20.4.3 Implementation 318
20.4.4 Test 330
20.5 Operating Phase 335
20.5.1 Deployment 335
20.5.2 Maintenance 337
20.6 Summary 337
20.7 Literature 338
20.8 Exercises 338

21 Runtime Environment 339
21.1 Introduction 339
21.2 Overview 340
21.3 Bytecode 341
21.4 Java Virtual Machine 344
21.4.1 Artificial Computer 344
21.4.2 Interpreter mode 345
21.4.3 JIT compiler mode 346
21.4.4 Hotspot Mode 346
21.4.5 Garbage Collector 347
21.5 Libraries 347
21.5.1 Native Libraries 347
21.5.2 Class Libraries 348
21.5.3 Resources And Property Files 348
21.6 Portability 349
21.6.1 Binary Compatible Bytecode 349

21.6.2 Porting Prerequisites ... 349
21.7 Program Start ... 350
21.7.1 Start Script ... 350
21.7.2 Native Wrapper ... 351
21.8 JVM Configuration ... 353
21.9 Summary ... 353
21.10 Literature ... 354
21.11 Exercises ... 355

22 Class Libraries ... 357
22.1 Introduction ... 357
22.2 Overview ... 358
22.2.1 Areas of Application ... 359
22.2.2 Reuse ... 359
22.2.3 Documentation ... 359
22.2.4 Language Extension ... 359
22.2.5 Types of Class Libraries ... 359
22.3 Java Standard Edition ... 359
22.3.1 Base Classes ... 360
22.3.2 Class »System« ... 368
22.3.3 Threads ... 371
22.3.4 Streams ... 372
22.3.5 Properties ... 374
22.3.6 Container Classes ... 376
22.3.7 Abstract Windowing Toolkit ... 377
22.3.8 Swing ... 386
22.3.9 JavaBeans ... 389
22.3.10 Applets ... 390
22.3.11 Java Database Connectivity (JDBC) ... 390
22.3.12 Java Native Interface ... 391
22.3.13 Remote Method Invocation ... 392
22.4 Java Enterprise Edition ... 393
22.4.1 Entity Beans ... 394
22.4.2 Session Beans ... 394
22.4.3 Message Driven Beans ... 394
22.4.4 Interfaces ... 394
22.5 Java Micro Edition ... 395
22.6 External Class Libraries ... 396

22.6.1 Apache Software Foundation ... 396
22.6.2 Eclipse Community ... 396
22.6.3 SourceForge ... 396
22.6.4 Other Open-Source Software ... 396
22.6.5 Commercial Software ... 397
22.7 Summary ... 397
22.8 Literature ... 398
22.9 Exercises ... 398

23 Rules ... 399
23.1 Introduction ... 399
23.2 Overview ... 400
23.3 Writing Conventions ... 401
23.4 Access Protection ... 402
23.4.1 Four Levels of Access Protection ... 402
23.4.2 Access Level »private« ... 403
23.4.3 Access Level »default« ... 403
23.4.4 Access Level »protected« ... 403
23.4.5 Access Level »public« ... 403
23.4.6 Case Study ... 403
23.4.7 Scope of Variables ... 409
23.5 Evaluation Order ... 413
23.5.1 Dot Before Dash ... 414
23.5.2 Dot Before Dot ... 415
23.6 Type Conversion ... 417
23.6.1 Implicit Conversion ... 417
23.6.2 Explicit Conversion ... 419
23.7 Polymorphism ... 422
23.7.1 Method Overloading ... 422
23.7.2 Method Overriding ... 424
23.8 Summary ... 428
23.9 Literature ... 428
23.10 Exercises ... 429

24 Algorithms ... 431
24.1 Introduction ... 431
24.2 Overview ... 432
24.2.1 Developing Algorithms ... 432
24.2.2 Types of Algorithms ... 433

24.2.3 Use of Algorithms ... 433
24.3 Develop Algorithms ... 433
24.3.1 Sorting Algorithms ... 433
24.3.2 Graphics Algorithms ... 434
24.4 Algorithm Usage ... 442
24.4.1 Sorting Algorithms ... 442
24.4.2 Search Algorithms ... 444
24.5 Summary ... 445
24.6 Literature ... 445
24.7 Exercises ... 446

Part IV Java Projects ... 447

25 Swing Programs ... 449
25.1 Introduction ... 449
25.2 Requirements ... 450
25.3 Analysis and Design ... 450
25.3.1 User Interface ... 451
25.3.2 Program Logic ... 451
25.4 Implementation ... 454
25.4.1 Start Eclipse With the Workspace »Exercises« ... 454
25.4.2 Create New Java project »Swing Programs« ... 455
25.4.3 Create New class »CourseStatisticsApp« ... 455
25.4.4 Implement Class »CourseStatisticsApp« ... 455
25.4.5 Create New Class »MainWindow« ... 456
25.4.6 Implementing Class »MainWindow« ... 457
25.4.7 Implementing Class »CsvParser« ... 468
25.4.8 Implementing the Class »TableFilter« ... 469
25.5 Test ... 472
25.6 Deployment ... 472
25.7 Summary ... 474

Part V Appendix ... 475

26 Frequent Errors ... 477
26.1 Introduction ... 477
26.2 Java Errors ... 477
26.2.1 Cannot Make a Static Reference To The Non-Static Field ... 477

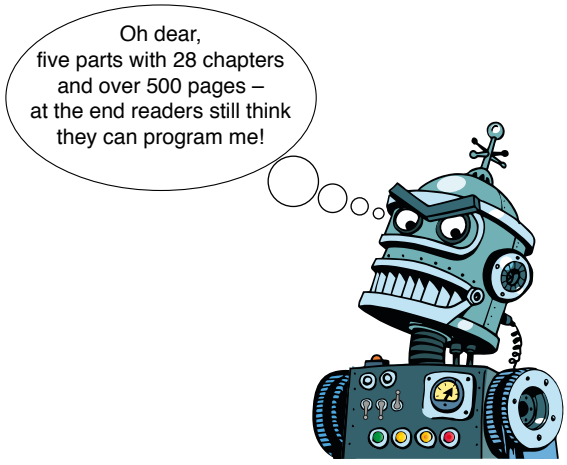
26.2.2 Output of a »null« Value 478
26.2.3 NullPointerException 479
26.2.4 Missing Break in Case Statement 481
26.2.5 Incorrect Comparison 482
26.2.6 Unhandled Exceptions 485
26.2.7 NoClassDefFoundError 486
26.2.8 ClassNotFoundException 486
26.3 Eclipse Errors 486
26.3.1 Eclipse Could Not Be Started 486
26.3.2 Chaotic Eclipse Perspective 486
26.3.3 Missing Window 487
26.4 Summary 487
26.5 Literature 487

27 Glossary 489

Index 491

Preface

Java is currently undisputedly the most important programming language. Therefore, many would like to learn Java. Unfortunately, getting started is not easy, because programming with Java requires at least two things: mastering the programming language *and* mastering a development environment. This is the reason why this book was written. With the help of many examples, it shows how the language is structured. In addition, the book uses the example of the Eclipse development environment to teach you how to develop Java programs with this tool.



Robert from the machine world accompanies you through the book.

The first part »Basics« gives you the Java and Eclipse basic knowledge. This part lays the programming foundations, gives you an overview of Java technology, and shows you what is special about object-oriented programming. A chapter about the Eclipse development environment completes this part.

In the second part »Java Language« everything revolves around the subtleties of the Java language. This is where the first small Java applications are created. This part offers a mixture of knowledge part and practical exercises. At the end of each chapter, you will find tasks that you can do on your own. With the solutions to the tasks at the end of this book you check the learning success.

Java technology is the focus of the third part »Java Technology«. Additionally, this part introduces you to the rules you must observe when programming, what class libraries are and what advantages they have. In addition, you will learn how to test programs, what algorithms are and how to program them.

A larger Java project is the focus of the fourth part. Here you will apply all the elements from the previous parts on an application with a graphical user interface. The project shows how to develop a larger application piece by piece with the Eclipse development environment.

The fifth part, »Appendix«, concludes this book with solutions to the tasks, with basics of information processing and a chapter on the most common mistakes that can occur when working with Eclipse.

The Plot

As a plot, I based the book on the (fictional) programming course »Java with Eclipse« taught by Professor Roth to four students. The programming course is accompanied by the robot named »Robert« and – among many others – mainly by these five characters throughout the book:

The programming course with Lukas, Anna, Professor Roth, Julia and Florian

Who is the book for?

This book is aimed at active readers. You don't want ready-made solutions, you want to program yourself. Without actively programming yourself and staying on the ball until your self-written program runs, you will not learn Java. The book contains a tempting number of ready programmed examples that you can run at the click of a button. Only reach for the sample solutions if you get stuck. First, try to enter the programs yourself and learn from the mistakes. Only by actively programming will you master Java and the Eclipse development environment.

Bonus Material

The book contains plenty of examples that can be easily imported into the Eclipse environment as solutions. You can easily download them from the Elektor publishing company homepage www.elektor.com/books/programming. Among these downloads, you will also find a bonus chapter, which is not printed due to space limitations. It explains the programming of so-called terminal programs.

Font Conventions

Various parts of the text are highlighted as follows for better readability:

Text Part	Meaning
Data types in body text	*Person*
Data types in headlines	»Person«
Keywords in body text	*implements*
Keywords in headlines	»implements«
Variables in body text	*roland*
Variables in headlines	»roland«
Window (graphical user interface)	ECLIPSE IDE LAUNCHER
GUI element (graphical user interface)	FINISH
Menu (graphical user interface)	FILE
Menu command (graphical user interface)	MENU → FILE → NEW → JAVA PROJECT
Files	*Samples.zip*
Directory paths	*C:/Programs/Eclipse*

Listing (source code from sample programs)	`1 package programmingcourse;`
	`2 public class Robot {`
	`3 (...)`
	`4 }`

Program output	`Result = true`
URL	*http://eclipse.org*
`(...)`	Due to space limitations part of the source code is missing

Acknowledgements

I would like to thank everyone who supported me in writing this book: the Elektor publishing company and my editor Ferdinand te Walvaart for his trust in my work and his great patience. I would like to thank the Hanser publishing company who has given permission to translate the original German manuscript.

As always, my wife Christiane has been very supportive of this project. Many thanks for your help! I would also like to thank Alina Neacsu, who cleaned my book manuscript of spelling mistakes. Many thanks also to Valeriy Kachaev (Studiostoks), from whom the templates of the robot cartoons come.

How to Contact Us

Despite the greatest care, it is not always possible to avoid overlooking one or another error in a book. If you find errors, have suggestions for improvement, or questions, just send me an e-mail at *java-eclipse@steppan.net*. I will answer your questions as soon as possible and try to include your suggestions for improvement in upcoming editions. You can find the most recent additions and further information at *http://www.programmingcourse.net*. Now I hope you enjoy reading and developing your Java programs with Eclipse!

Bernhard Steppan Wiesbaden, March 2023

PART I

Basics

To be able to develop computer programs, you need to master the basics. Chapter »Programming Basics« lays the groundwork for programming Java applications. In chapter 2, »Technology Overview«, you will learn what Java has in common with other programming languages and how Java differs from other languages.

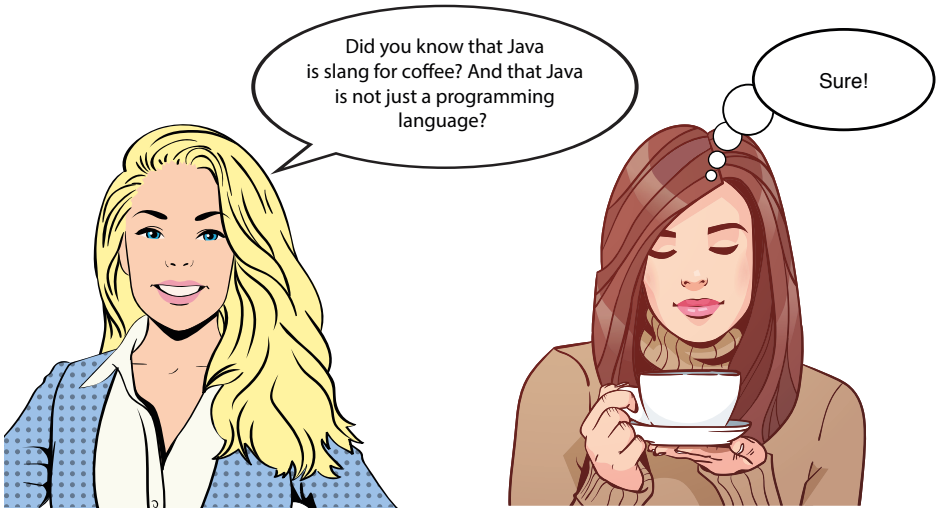


Figure 1: To develop computer programs, you need to master the basics.

Afterwards, it continues with the chapter »Object-Oriented Programming«. It shows what is special about object-oriented programming and how object-oriented programs are structured. The chapter »Development Environment« concludes this part of the book with the installation of the development environment and introduction to »Eclipse«.

1 Programming Basics

1.1 Introduction

Programming means writing computer programs. Computer programs consist of one or more commands in a programming language. The robot named Robert presents a simple Java program to the students in Professor Roth's programming course (Figure 1.1).

```
class Hello {
   public static void main(String[] args) {
      System.out.print("Hello!");
  }
}
```

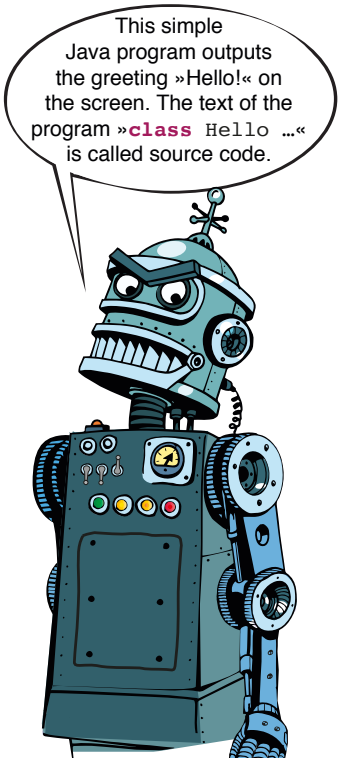


Figure 1.1: Robert from the machine world is *the* expert for machine programs.

The students of the programming course think that there are a lot of instructions for such a simple program. Anna would like to know from Professor Roth whether it could be simpler:

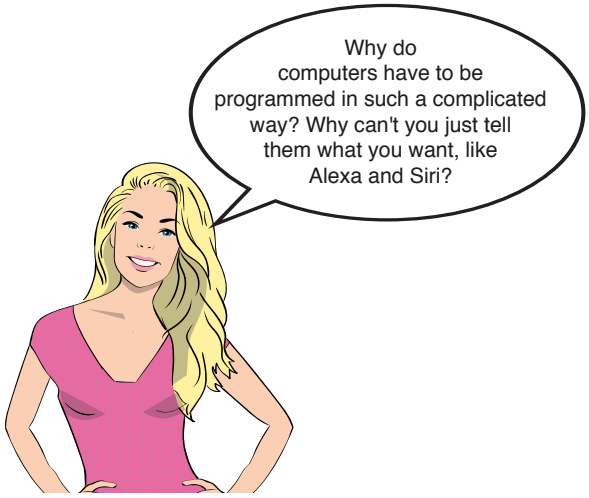


Figure 1.2: Is it really still appropriate today to type in programs?

»Even Alexa, Cortana and Siri«, says Professor Roth, »are just computer programs.« These programs were developed so that humans can control computers via speech. But Alexa & Co. can only do the few tasks for which they were specially programmed. If you want the computer to perform other tasks, such as word processing, you have to write a special program for it. These programs can be developed in Java, for example.

1.2 The Language of the Machine World

When we speak of computers today, we always mean *digital* computers. These machines understand only their digital machine language. Digital means that the computer uses binary code for all information. Therefore, the computer's machine language consists only of a sequence of zeros and ones.

However, the computer's machine language, with its sequences of zeros and ones, is extremely difficult for humans to understand. To program computers directly in machine code would therefore be completely absurd. It would take a very long time and the probability of errors would be high.

If you want to program the computer close to the machine, you use an auxiliary language. This auxiliary language is called assembler language or assembler for short. Professor Roth presents a simple example to his programming class. Like the Java program before, it simply outputs the character string »Hello!« on the screen (Figure 1.3).

Professor Roth's programming course finds the assembly language program quite difficult to understand. How could only programmers learn such a terrible language? The answer is simple, because, in the early days of computers there were no high-level languages like

Java. Programmers had to pay close attention to the computer's processors when they programmed the machine in assembly language.

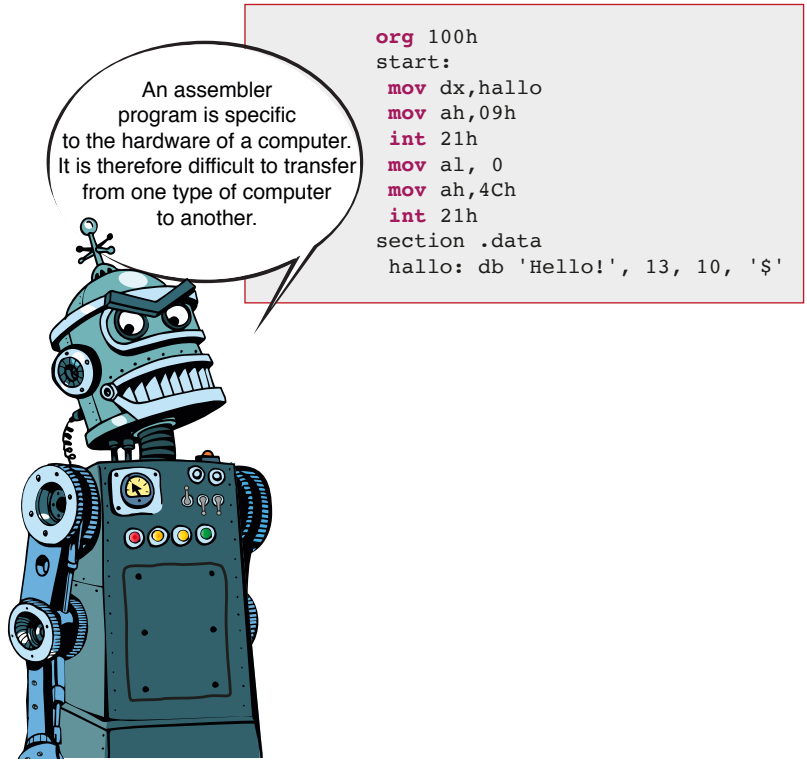


Figure 1.3: This assembler program also outputs »Hello!«.

Assembler programs are much longer compared to functionally equivalent Java programs. They consist of many small-part commands which, taken alone, do little. Therefore, one needs many of these commands to write a larger program. This program is written specifically for one type of computer. It is difficult to transfer to another type of computer.

Besides the high cost of developing such programs, a major disadvantage of the assembly language is that it is difficult to transfer from one type of computer to another. However, the small-part instructions do not have only disadvantages. They have the advantage that a good programmer can use them to create very lean and fast machine programs. They also often require far less main memory than comparable programs written in a high-level language.

1.3 High-Level Programming Languages

It seems to be somehow bewitched: Computers only understand their specific machine language. We, on the other hand, without special programming training, understand only

our native language and maybe one or two foreign languages. How can we bridge this huge gap between the machine world and the human world?

We can either develop even more powerful programs like Alexa, Cortana and Siri, so that the computers execute everything we want. Or we can learn the computer's language if we want to develop special programs for tasks that Alexa & Co. can't do – no, those aren't the only options, because there is, fortunately, a third way.

Programming a complex program in assembler is no longer up-to-date. That's why people started very early to develop programming languages like Java. These languages form a bridge between the (for most humans) difficult to understand machine language and the (for most machines) difficult to understand human language. These languages are called high-level programming languages or high-level languages for short.

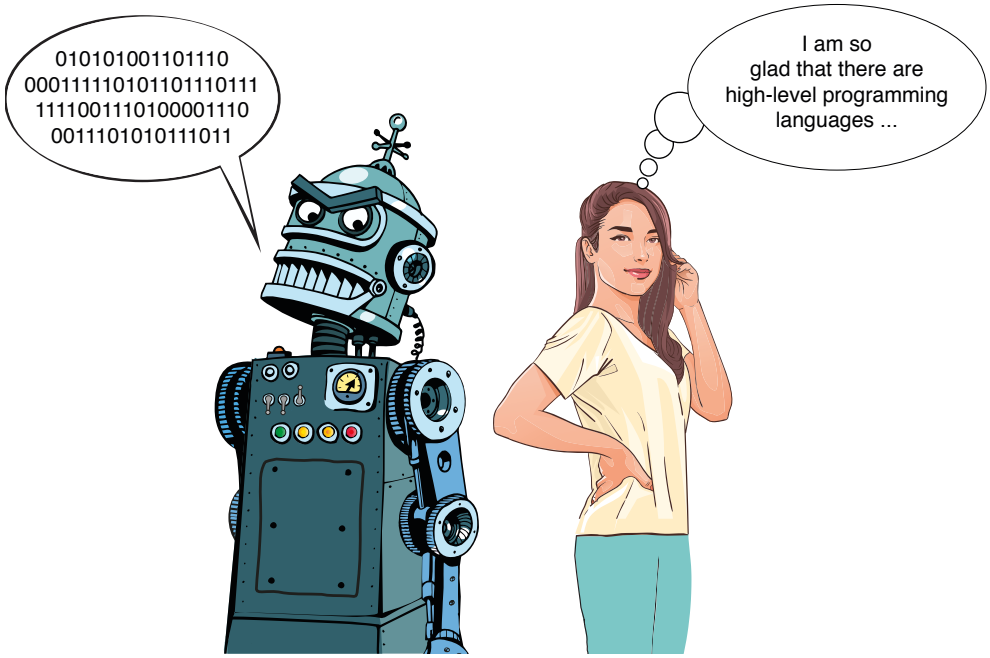


Figure 1.4: High-level programming languages are mediators between humans and machines.

High-level programming languages are much easier for a human to learn and understand than the language of the machine world. But how does it work? How do you translate a high-level language into the language of the machine world? For this purpose, a trick has been thought of. This trick is a special program that translates the source code of a high-level language like Java into the language of the machine world. This program is called a compiler and is part of a development environment.

1.4 Development Environment

1.4.1 Compiler

The compiler is one of the core components of a development environment like Eclipse. It transfers the source code of a Java program into the language of the machine world. The source code is the text that was seen in figures 1.1 and 1.3.

1.4.2 Editor

In the editor you enter the source code of a program as in a word processor. An editor also provides program development support, such as advice on how to fix the errors that are displayed.

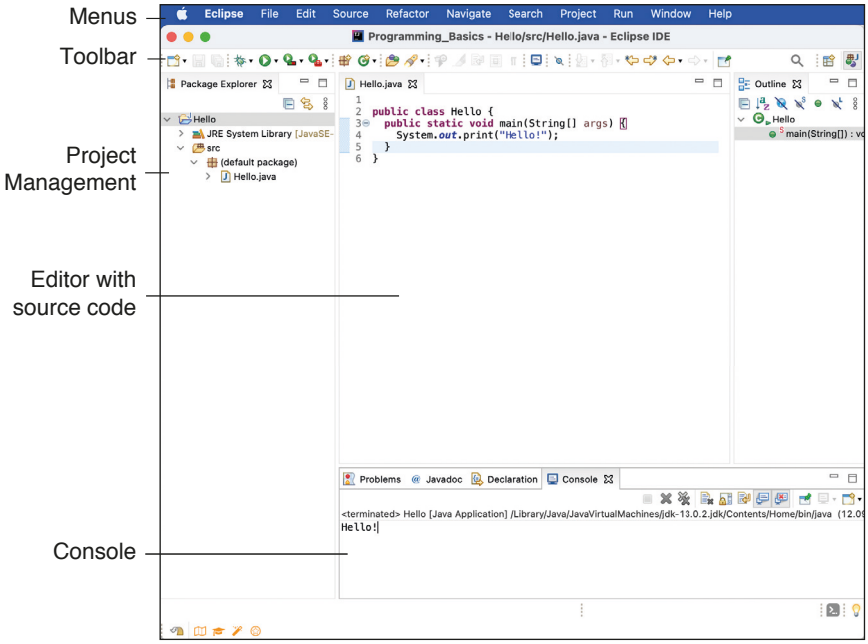


Figure 1.5: Editor, compiler and project management of the Eclipse development environment.

1.4.3 Project Management

Java programs usually consist of a large number of files. So that you do not lose the overview, the Java development environment has a project management. It shows which files belong to a project.

1.5 Runtime Environment

Java programs require a special runtime environment. In other words: Java programs only run with an additional program on your computer. You don't notice this at first, because the Eclipse development environment calls this runtime environment in the background when you run a Java program. To understand what this runtime environment is all about, turn to the next chapter. It shows you how Java has evolved and why a Java runtime environment is necessary at all.

1.6 Summary

Programming means writing computer programs. Computer programs consist of one or more commands in a programming language. These commands are written in the form of a text. In programming, this text is called source code. Computers expect the commands in machine language. We, on the other hand, speak in our human language. To bridge this gap, computer scientists have developed high-level languages. Java is one of these high-level languages.

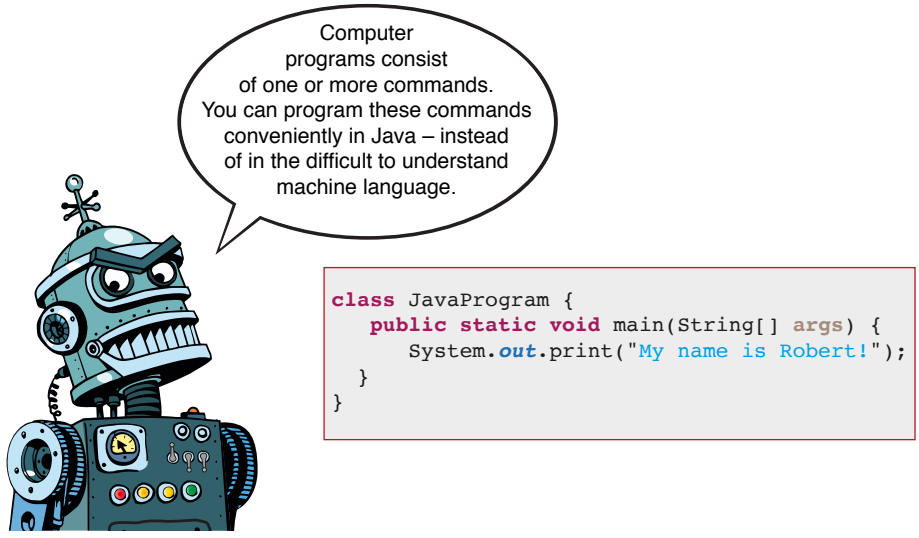


Figure 1.6: Computer programs consist of commands of a programming language.

To translate a high-level language program into machine language so that the computer can execute it, you need an additional program. This translation program is called a compiler. The compiler is part of the development environment. This consists (among other things) still of an editor and project management. With the help of the editor, you write the source code of a program. The project administration administers the different files, which belong to a project.

1.7 Literature

Bernhard Steppan: »A Brief History of Programming Languages«;
https://www.programmingcourse.net/articles/a_brief_history_of_programming_languages

1.8 Exercises

To deepen your knowledge, please go to *https://www.programmingcourse.net/books/java_with_eclipse/programming_basics* and work on the exercises listed there. There you will also find the solutions to the exercises.

2 Technology Overview

2.1 Introduction

Java is more than a successful programming language. Java is a technology. This chapter gives you an overview of the components of this technology and shows you what makes Java stand out from other programming languages and technologies.

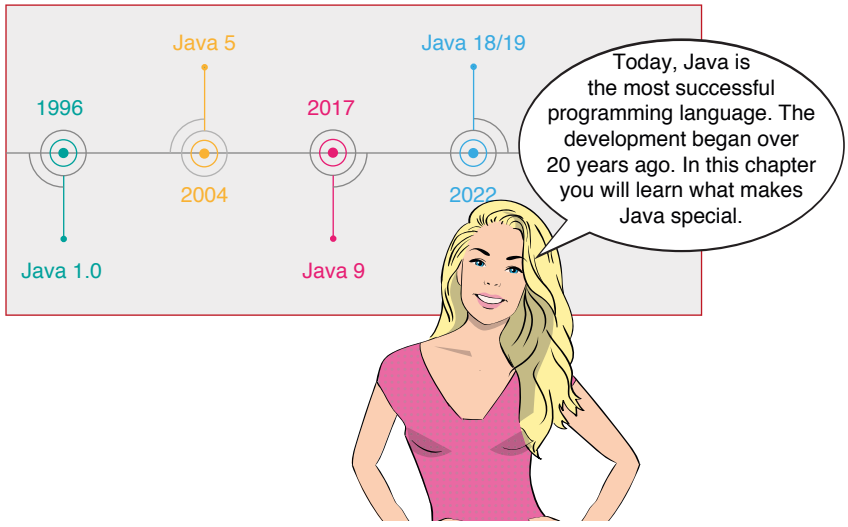


Figure 2.1: Anna gives you an overview of the Java technology.

2.2 Overview

2.2.1 The Early Days of Java

The history of Java began when some programmers from the Californian company Sun Microsystems were asked to develop an object-oriented programming language called Oak (Object Application Kernel) for programming household appliances in 1991. Household appliances are mostly not very powerful. Therefore, Oak programs should be compact so that they can run quickly. Since home appliances use different software and hardware, Oak programs should be as easy as possible to transfer from one appliance to another. It turned out that these requirements were also an ideal fit for implementing Internet programs. Without further ado, the Oak team was given the task of developing the first Internet browser for Oak programs.

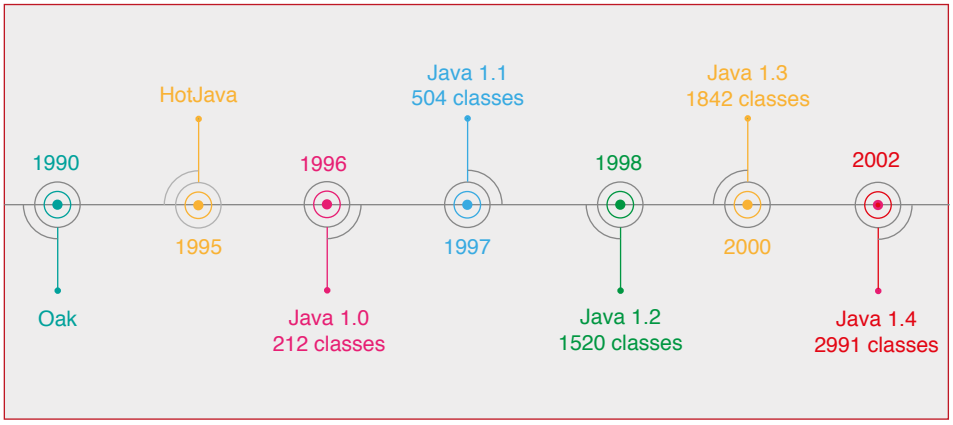


Figure 2.2: The early days of Java.

About a year later, the new browser could display small programs (applets) on HTML pages. It saw the light of day as »HotJava« with the programming language Java[1] in 1995. Java 1.0 followed in 1996 and was replaced by Java 1.1 in the same year. Java programs were still very slow at that time. In addition, programming graphical user interfaces (GUIs) was difficult at the beginning. This only ended with Java 1.2. This version brought a new GUI class library called Swing. The increased performance of Java was also expressed in the size of the Java class libraries. The number of classes increased explosively to over 1500 classes – Java became a technology.

In 2000, Java 1.3 appeared with the so-called hotspot optimization. It translated frequently used parts of a program (hotspots) directly into native machine code. This led to Java programs running significantly faster for the first time. The successor Java 1.4 brought some language extensions and the introduction of Java Webstart. Java Webstart updates a Java program automatically at program start, provided that updates are available for this program.

[1] Java is a slang expression for coffee. According to an anecdote, the team that developed Java named the new programming language after a type of coffee.

2.2.2 The Growth Period of Java

With the successor Java 5, the extent of the class libraries did not increase as strongly as with Java 1.4. The changes to the programming language had it however in itself: Java 5 was characterized by so many serious changes, as there were before only with Java 1.2. Sun Microsystems has thoroughly improved the Java language with version 5. In addition, the Java inventor introduced the product number – the leading one was omitted. Among other things, the product number is supposed to indicate the maturity and stability of the Java version. Internally, Java 5 continues to be referred to as version 1.5. Also all following Java versions carry internally still the old version designations.

Java 6 in the year 2006 brought above all again a further improvement of the execution speed. A further caesura was that Sun Microsystems split off the later OpenJDK from this version (JDK = Java Development Kit).

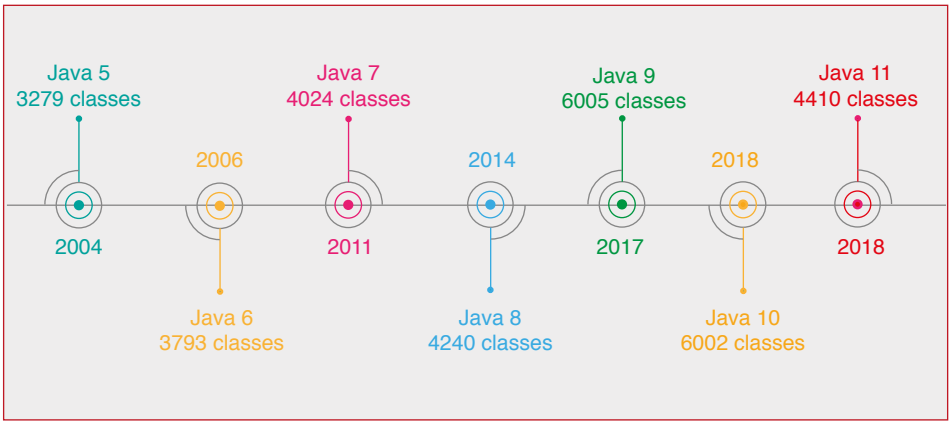


Figure 2.3 The growth period of Java

In 2010 Oracle took over the Java inventor Sun Microsystems. This brought a lot of turmoil to the Californian company, stalling the progress of Java development. It was not until the summer of 2011 that the Java developers reported back with the new version 7. One of the biggest innovations of this version was the integration of the GUI class library JavaFX. Sun Microsystems introduced this class library earlier as an alternative to the GUI class library Swing. The integration of the new library and other classes resulted in the number of classes shipped with Java breaking the sound barrier of 4000 for the first time.

With Java 8 came further language improvements and the so-called LTS version. LTS stands for »Long Term Support«. This is a version with support from the Java manufacturer for a longer period of time. This is especially important for companies that use Java and need support in case of errors for their business-critical Java programs. Another innovation in Java 8 was the integration of a JavaScript Runtime Environment. This made it possible to execute JavaScript directly in Java applications.

The successor Java 9 appeared with considerable delays in 2017. The reason was long-lasting discussions about the new module system with the code name Jigsaw. The module system should allow the same classes to be integrated in different versions in a Java application. Java Webstart was marked as »obsolete« in this version. A class or function that was

marked as »obsolete« could be removed from the Java base at any time. Otherwise, the size of the class libraries increased again by over 1700 classes.

With Java 10 Oracle changed its Java course and began to »clean up« the Java class libraries somewhat. In addition, Oracle began to publish two Java versions every year with version 10. With version 11, Oracle again offers a version with Long Term Support (LTS). The Java manufacturer cleaned up the successor version Java 11 even more extensively. This had the consequence that Applets, Java Webstart and unfortunately also the GUI library JavaFX disappeared from the Java base. These huge changes to the last Java versions meant that many companies did not introduce these newer Java versions.

2.2.3 The Presence And Future of Java

Java 12 brought among other things switch expressions as preview. They extend the previous switch statements and are a fixed part of the language from Java 14. Java 15 introduced, among other things, so-called sealed classes and interfaces as a preview. Sealed classes and interfaces restrict other class to implement or extend them.

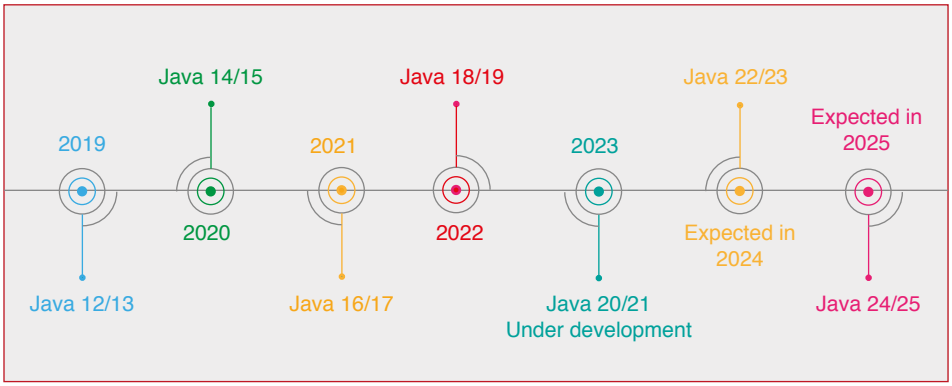


Figure 2.4 The presence and future of Java

Java 16 brought a number of internal changes. This affected for example the source code of the JDK, which is written in C++. With Java 16 it is allowed to use C++ 14 language features. With Java 17, Oracle again offered a version with Long Term Support. Another important innovation of this version was the support for Apple's new computer architecture. For this there is a so-called MacOS/AArch64 port of the JDK.

With Java 18, Oracle introduced UTF-8 as the default character set for the standard Java APIs. At the time this book went to press, Java 19 had been released. With this version a port of the JDK for Linux/RISC-V will be available. Another important innovation of this Java version is virtual threads. They allow a programmer to develop systems that require comparatively few resources from the operating system even under heavy load.

2.3 Why Java?

Java is one of the most successful programming technologies today: it is used at many universities and companies. Many private users also program in Java. The reasons lie in the features of the Java technology.

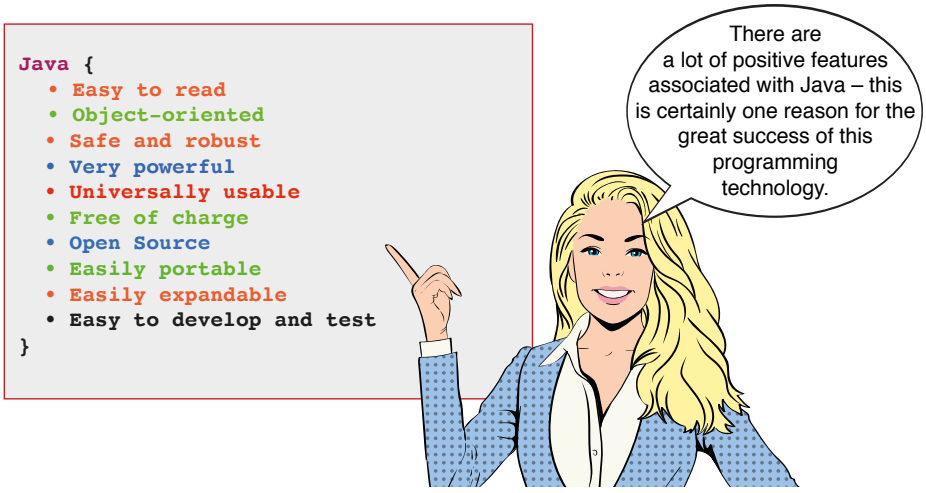


Figure 2.5: The features of Java technology.

2.3.1 Easy to Read

The simple syntax of Java has the advantage that you can learn the language comparatively easily and read Java programs easily. The simple syntax also means that you can more easily understand the structure and meaning of Java programs that you have not developed yourself.

2.3.2 Object-Oriented

The fact that Java programs are easy to read is not only due to the syntax of the programming language, but also because they are structured in an object-oriented way. Object orientation is one of the most important reasons for the clear structure of Java programs. In the chapter 3, »Object-Oriented Programming«, you will learn everything about the topic.

2.3.3 Safe And Robust

One of the most important features of the Java programming language is that it supports the development of safe and robust programs. Java programs owe part of their robustness

to their object orientation. To make Java programs run more stably than object-oriented C++ programs, the developers of the Java language have removed error-prone C++ constructs. This has ensured that Java programs generally run much more safely and do not crash very easily, as some poorly developed and tested C++ programs do.

2.3.4 Very Powerful

The Java programming language helps you develop programs that can meet the most demanding requirements for commercial applications. With such professionally developed Java programs, thousands of users can work efficiently in parallel, safely and without interruption.

2.3.5 Universally Useable

There are programming languages specifically for scientists, others specifically for business people. Some are designed specifically for programming graphical interfaces, others specifically for database queries. Java is so successful because the language can be used universally. The spectrum of Java begins with small programs for private use, continues with tools such as the Eclipse development environment and extends to web applications for large companies. Java can even be used for such exotic programs as controlling Lego robots.

2.3.6 Free of Charge

Another advantage of Java is that the technology has been free of charge since its beginnings. Even professional Java programs can be developed for free.

2.3.7 Open Source

The Java inventor Sun Microsystems has made Java open as OpenJDK (JDK = Java Development Kit). This means that the source code and thus the know-how of the JDK is public. The technical term for this is open source. This ensures that the Java programming platform can be transferred to other operating systems independently of a specific manufacturer. Java programs can therefore be run on (almost) any computer system.

2.3.8 Easily Portable

Forget if you have read or heard somewhere that Java is platform independent. The well-known saying »Write once, run anywhere« is a nice fairy tale that Sun Microsystems came up with to help Java succeed. Large sections of IT management, many specialist authors and journalists still believe this story today.

The truth is: Java programs are comparatively easy to transfer (port) from one operating system to another. This is because Java programs can rely on finding almost the same conditions on other operating systems as on the operating system on which they were developed.

Table 2.1: Operating systems currently supported by Java (64 bit).

Java version	Operating system
`Java 13`	Linux x64, MacOS x64, Windows x64
`Java 14`	Linux x64, MacOS x64, Windows x64
`Java 15`	Linux AArch64, Linux x64, MacOS x64, Windows x64
`Java 16`	Linux AArch64, Linux x64, MacOS x64, Windows x64
`Java 17`	Linux AArch64, Linux x64, MacOS AArch64, MacOS x64, Windows x64
`Java 18`	Linux AArch64, Linux x64, MacOS AArch64, MacOS x64, Windows x64
`Java 19`	Linux AArch64, Linux x64, MacOS AArch64, MacOS x64, Windows x64

Java programs are extremely easy to port under two conditions: The first prerequisite is that the developers of a Java program have not used any specialties of an operating system that are not covered by Java. The second prerequisite is that a Java version suitable for the program exists for the target operating system. The first requirement is very easy to fulfill. The second prerequisite is usually given by the fact that Java is open source.

2.3.9 Easily Expandable

The term Java class library has come up several times before, without me explaining it in more detail: Java class libraries are pre-built program parts that your program can easily use. If you research on the Internet whether a certain problem is solved by a Java class library, you will be amazed how large the offer is.

One of the reasons why Java is so successful is because of the variety of Java class libraries. You do not have to develop all program parts yourself. Instead, you can draw from the pool of prefabricated and proven solutions. And the best thing is: like Java, many of these class libraries are open source and free of charge.

2.3.10 Easy to Develop And Test

In the early days of Java, developing Java programs was painstaking. There were simply no professional development and testing tools. That has changed dramatically. Part of Java's success is certainly due to the fact that there are hardly any other programming languages with as many professional development environments as Eclipse IDE or IntelliJ IDEA. The fact that many professional tools are also free of charge was another plus point for many companies.

2.4 What Belongs to Java?

On the previous pages the term "Java technology" was mentioned several times. I wanted to make clear with it that Java is far more than *only* a programming language in comparison with the predecessor languages C and C++. But what does Java technology consist of? It provides a uniform basis for the development, execution and operation of programs. Java technology is based on the pillars »Java Programming Language«, »Java Virtual Machine«,

»Java Class Libraries« and »Java Development Tools« (Figure 2.6).

If Java is your first programming language, you may be confused by the many new terms and abbreviations in this section. Be patient. You don't need to memorize everything, because throughout the book the terms are repeated with many examples. And after the first program examples, the meaning of the terms will become much clearer to you.

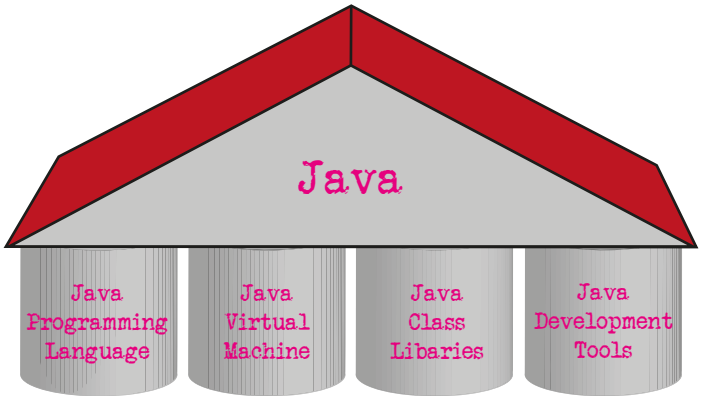


Figure 2.6: The Java technology rests on four pillars.

2.4.1 Java Programming Language

The Java programming language is the core of the Java technology. Like any programming language, Java has a special syntax to write a program. The developers of the Java programming language based their syntax on C++ to make the transition to Java easier. However, some elements, which I will discuss in more detail later, were greatly improved, resulting in the ease of reading and robustness of Java.

2.4.2 Java Virtual Machine

Java programs are not stored as executable files on Windows, for example. Therefore, you (usually) need a virtual machine to run the Java program. Behind the mysterious name »virtual machine« is a special program. This program executes Java programs. In the case of Java, the program is called »Java Virtual Machine«. Since the term is so long, it is usually abbreviated as Java VM or simply JVM.

This Java VM must be specially adapted for operating systems such as Windows, Linux or MacOS. In other words, there is a special Java VM for Windows, one for Linux and one for MacOS. This special Java VM is one of the prerequisites for easily transferring Java programs from one operating system such as Windows to another such as Linux (Figure 2.7).

So if you take it very strictly, Java programs are not platform-independent at all, as often claimed. On the contrary: they are platform-dependent. To be more precise: They are dependent on the Java platform together with the Java VM contained therein. So that Java programs can be transferred easily from one operating system to another, the Java inventor Sun Microsystems resorted to a trick. He simply developed a Java platform for each desired target operating system.

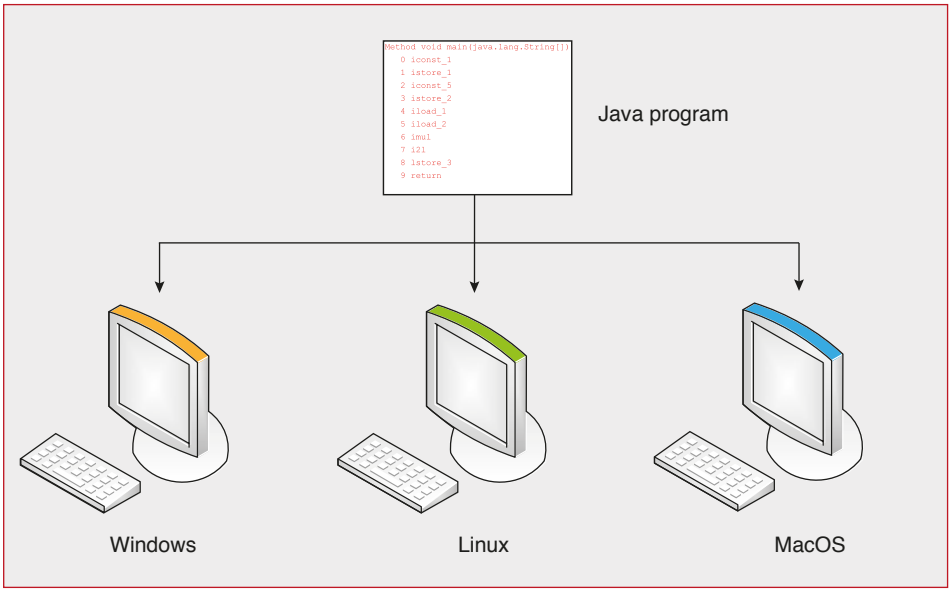


Figure 2.7: Java programs run on various operating systems using the Java VM.

This Java platform protects each Java program from the special peculiarities of each operating system. Now you may ask: Who develops this Java platform? In the early days of Java, the Java inventor Sun Microsystems did it themselves. Since Sun Microsystems was bought by Oracle, primarily Oracle takes care of these different Java ports. Since Java is open source, the further development of Java is increasingly taken care of by the open source community of programmers in addition to Oracle.

Only if a Java platform exists for a certain operating system environment, Java programs can be executed on this environment. This is also the reason why you have to install Java (and therefore a Java VM) on your computer operating system. Chapter 4, section 4.2.2, »Install Java«, shows you exactly how to do this.

2.4.3 Java Class Libraries

This term also came up several times before, without me explaining it in detail. Class libraries are prefabricated programming parts consisting of Java classes. What the term »class« means exactly, I will come back to later. Just this much at this point: class libraries make programming easier for you, because you can fall back on prefabricated program parts.

When you install Java, the most important Java class libraries are already included. These libraries are therefore included in the Java standard. Together with the Java VM, the class libraries form the Java runtime environment (Figure 2.8). Java runtime environment is abbreviated JRE.

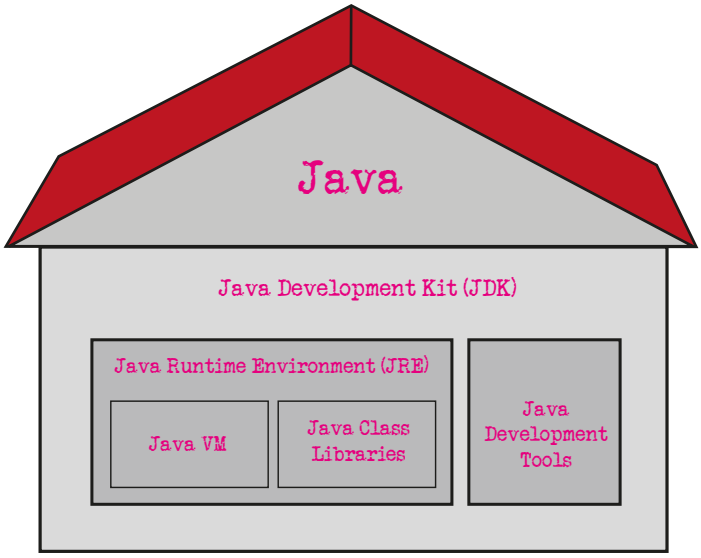


Figure 2.8: The Java Development Kit consists of tools and the runtime environment.

2.4.4 Java Development Tools

Java development tools form the fourth important pillar of the Java technology. The development tools provided with Java can be used to compile Java programs so that they can be executed by the Java Runtime Environment (JRE). In addition, there are other tools to test and deliver programs. Together with the JRE, the development tools form the Java Development Kit (JDK).

2.5 Java Versions

In the early days of Java, it took a relatively long time for a new Java version to be released (section 2.2, »Overview«). Meanwhile, new Java versions are released every six months. At the time this book went to press, Java 17 was just available, so this book is based on that Java version. However, most of the information in this book also applies to older Java versions back to Java 8 and – with high probability – to newer versions as well.

2.6 Java Editions

Java inventor Sun Microsystems originally released three editions of the Java platform. For this book only the Java Standard Edition (Java SE) is important. For the sake of completeness, however, I would like to present all three editions here in a comparative way.

2.6.1 Java Standard Edition

The simplest edition is the Java Standard Edition (Java SE or JSE). With this edition you can develop simple Java programs as presented in this book. You can equate this edition with the Java Development Kit (JDK) in a simplified way.

2.6.2 Java Enterprise Edition

Java Enterprise Edition (Java EE or JEE) is an extension of the Java Standard Edition. Java creator Sun Microsystems released this edition to better support enterprise application development using Java. JEE, however, is not a product like the JDK that can be downloaded anywhere, but rather »just« a set of specifications from the Java inventor. Open source developers can use these specifications to program class libraries for enterprise applications. Of course, you can download them again if you plan to develop such an application.

2.6.3 Java Micro Edition

This Java Micro Edition (Java ME or JME) has its roots in the origins of Java. As you may recall, Java was intended to be developed to better program household appliances. Since these devices are not very powerful, Sun Microsystems developed a particularly small edition of the Java programming platform. Since every smartphone today is so much more powerful than the household appliances of that time, this edition has become more or less obsolete.

2.7 Summary

Java originated in 1995 as the Oak programming language for home appliances. The following year, 1996, Sun Microsystems released the first version of Java. Sun Microsystems designed Java as a technology. This Java technology consists of the Java programming language, the Java class libraries, the Java Runtime Environment (JRE) and the various development tools.

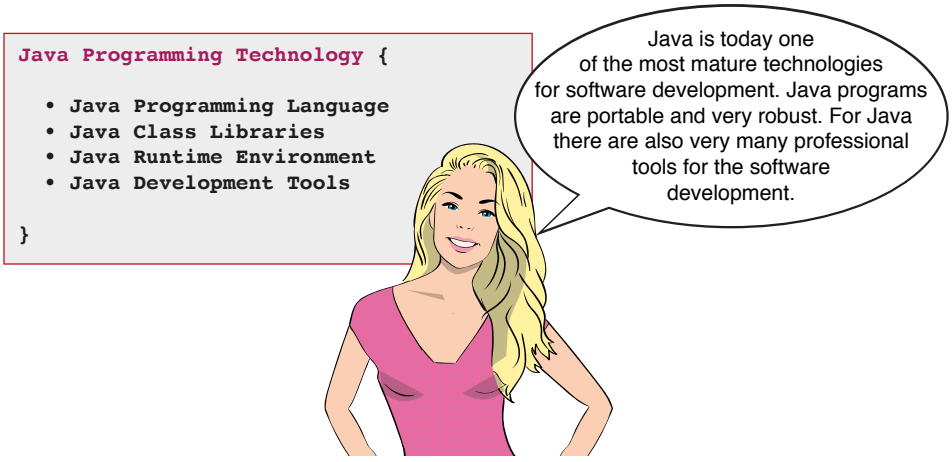


Figure 2.9 Java programs are easily portable and very robust.

The Java Runtime Environment runs Java programs. It consists of the Java Virtual Machine (JVM) and the Java class libraries. Each operating system requires its own runtime environment. Java programs can be developed and tested with the Java development tools. The Java Runtime Environment and the Java development tools together are called Java Development Kit (JDK).

2.8 Literature

Bernhard Steppan: »Oracle Slims Down Java 11«;
https://www.programmingcourse.net/publications/articles/oracle-slims-down-java-11 Supported operating systems and processors: *https://jdk.java.net/archive*

2.9 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/technology_overview
There you will also find the solutions to the exercises.

3 Object-Oriented Programming

3.1 Introduction

Java is an object-oriented programming language. Therefore, it is important to understand exactly how object-oriented programming works. This chapter is all about objects and classes, attributes and methods, and how objects can be protected from encroachment by other hostile objects. You'll also learn why object orientation came into being and what's so special about it.

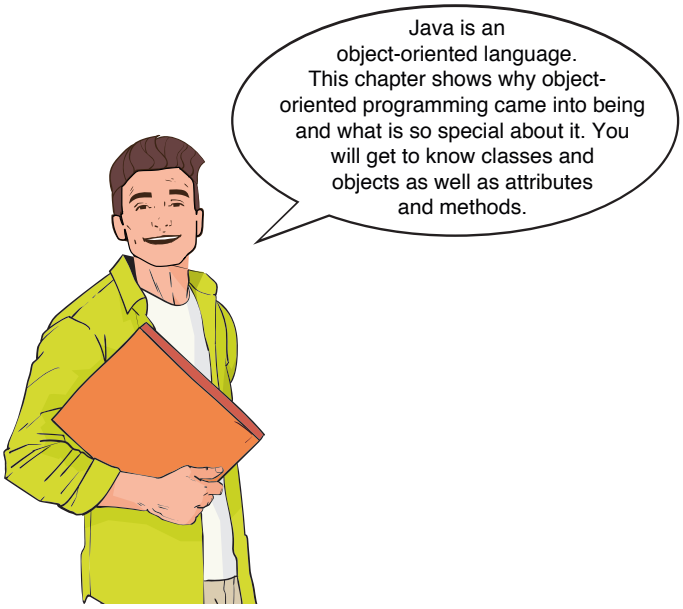


Figure 3.1: Florian explains what is special about object-oriented programming.

3.2 Overview

How did it all come about? It all started in the mid-60s of the 20th century. At that time, there was a software crisis. It was triggered by the increased demands on computer programs. As a result, the software became more complex and more buggy. At congresses, experts discussed the causes of the crisis and the reasons for the increased error rate.

A part of the software experts came to the conclusion that the software crisis could not be mastered with the conventional programming languages. They criticized at the conventional programming languages above all that the natural world could be represented so far only inadequately. They therefore began to develop a generation of new programming languages.

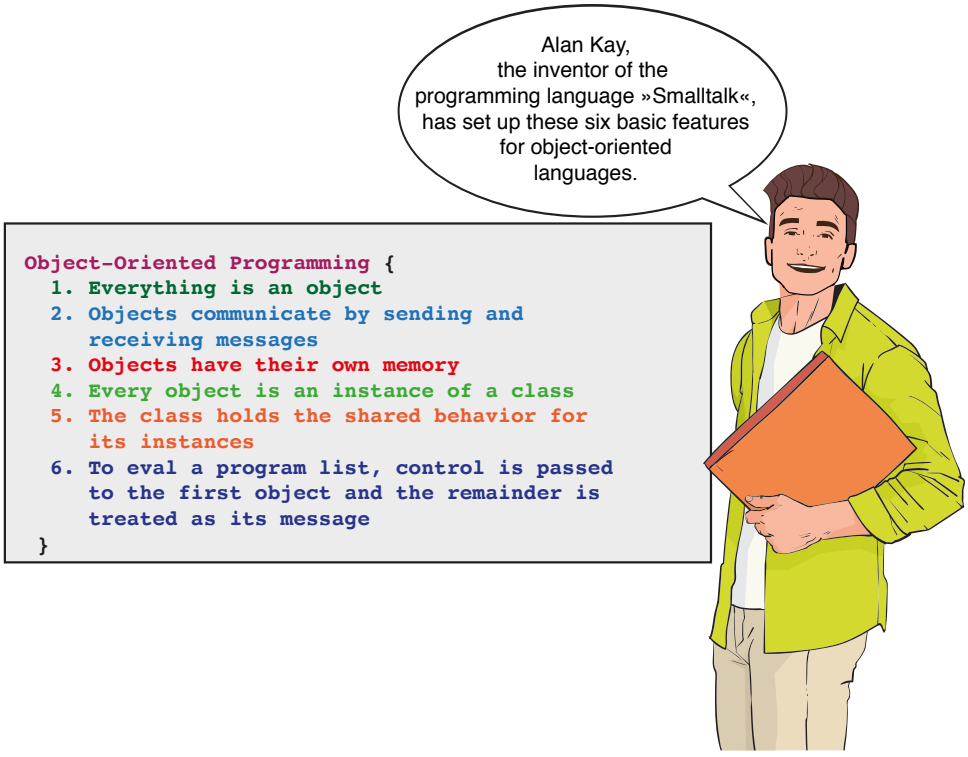


Figure 3.2: Features of object-oriented languages.

The experts began to use natural terms from the form theory of classical Greek philosophy for the new programming languages. They transformed these terms for programming (Figure 3.2). Since everything revolved around the notion of object, they called the new generation of languages »object-oriented«.

3.3 Object

Objects are to a Java program what cells are to an organism: A Java program is composed of these smallest units. If you look at a set of similar objects, you will notice that their basic *appearance* is common. In object-oriented programming, such objects are often referred to as *instances* of a class.

As an example again the programming course of professor Roth is to serve. Anna, Julia, Lukas and Florian take part in the programming course. Let us first pick out only the students Anna and Julia. Both students are objects with many similarities. For example, they have in common that they are women, attend the same programming course and are enrolled at the same university.

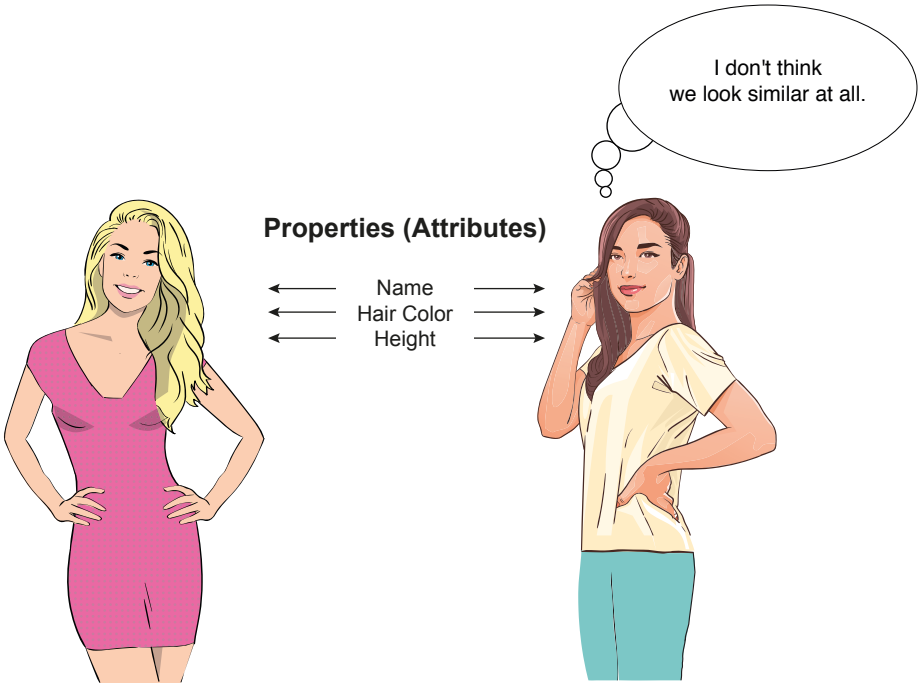


Figure 3.3: The »objects« Anna and Julia are similar.

From the point of view of object-oriented programming, this means that these two objects are *similar*. The differences between these objects result from the different value of their *attributes*. For example, both female students have a different name and different hair color as well as size (Figure 3.3).

Thus, similar objects have only their *principle* shape and certain capabilities in common. Everything else is *individual*. The common shape and the common capabilities of objects are determined by the building plan of the objects. This blueprint is called *class* in object-oriented programming.

3.4 Class

It is the *class* that shapes the principal form and capabilities of objects like the two female students. A class relates to an object as the blueprint of a human being relates to a real human being. The class gives an upper hand to different objects of the same kind. It is also said that a class *classifies* its objects – hence the name.

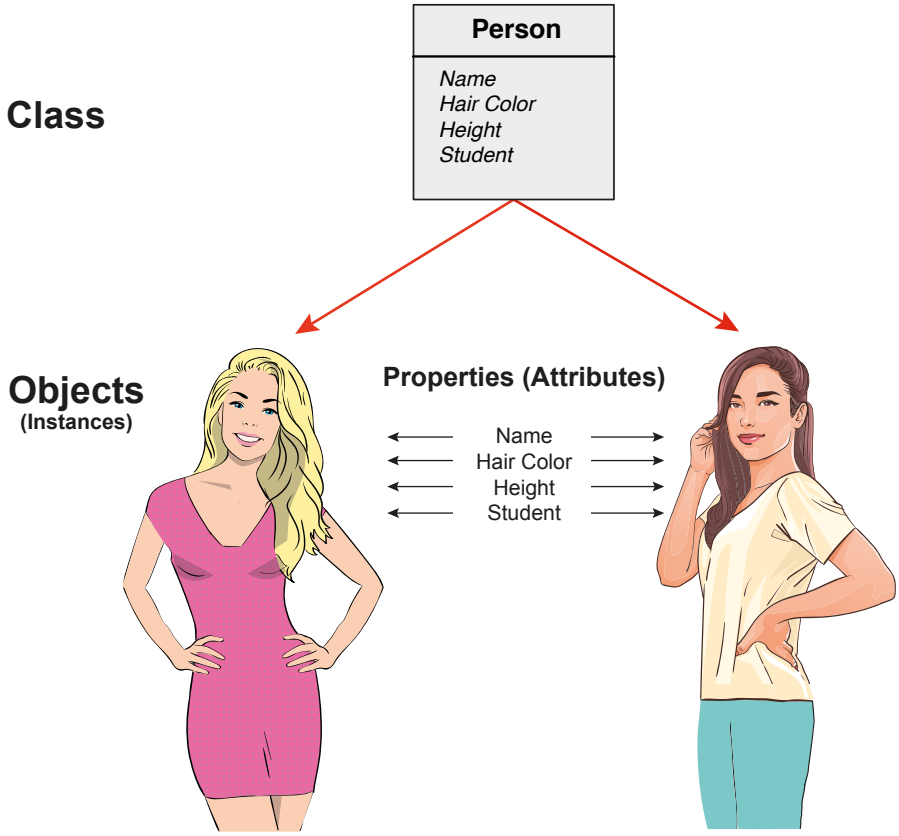


Figure 3.4: The class »Person« provides the blueprint for the »objects« Anna and Julia.

3.4.1 Properties

Our new class *Person* should get the attributes *name, hair color, height* and additionally *student*. Attributes are also called properties. You use these properties to specify the characteristics of an object. When new person objects are created from this class, all instances have an individual *name*, an individual *hair color*, an individual *height*, and an individual value for the *student* property (Figure 3.4).

Constants and Variables

For example, *Anna* should have the following properties: name = Anna, hair color = blonde, height = 1.71 m. Her girl friend from the programming class is named Julia, has hair color = brown, and is 1.72 m tall. Although both persons have been created according to the same blueprint (class), two clearly individual objects have been created: Both have different names, have different hair color, and are both different heights. Some properties, such as height, could be considered invariant, others more mutable. A special form of the variable property of an object is its state.

States

Some properties of the two individuals have been assigned fixed values, while others have been assigned variable values. Unlike the fixed properties, the flexible properties describe the *state* of the object. For example, the *student* property describes whether a person is currently enrolled at a university. The state of an object can change over time.

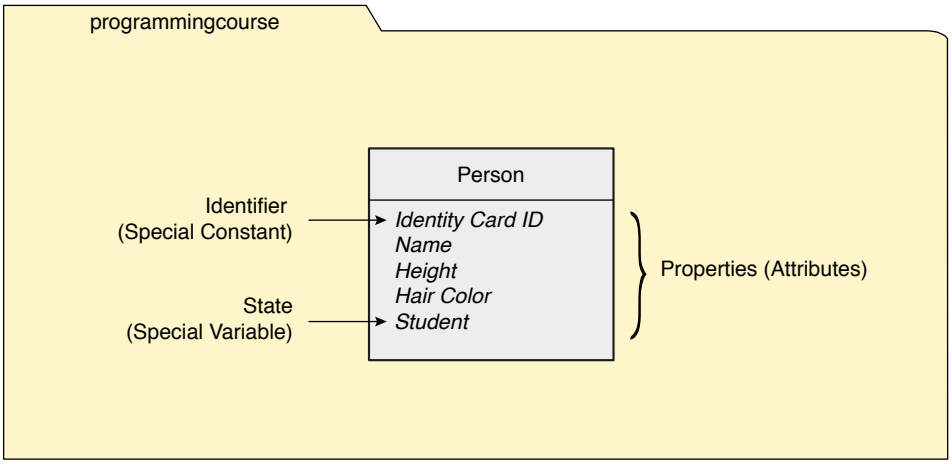


Figure 3.5: Constants, variables, states, and identifiers are properties of a class.

3.4.1.1 Identifier

What would happen if you created the objects *Anna* and *Julia* so that they had the same *size*, the same *hair color*, and the same state *student*? How could they be distinguished then? In this case, both objects have been given individual values for their attributes, but they happen to be the same. Thus, objects in a program are also alike, like identical twins.

So in order to distinguish objects better, you need something like a genetic fingerprint. In programming, the developer assigns a so-called identifier. This identifier is an additional attribute for which care is taken that it is *unique*. Only the identifier of an object ensures that the program can distinguish different copies even if their attributes happen to have the same values.

3.4.2 Methods

Suppose you want to tell *Anna* to respond to a question. In real life, you simply ask Anna a question or send her a message. In object-oriented programming, you call a method of the object *Anna* instead (Figure 3.6). A method is the object-oriented term for a function. The term was introduced to express that an object-oriented function is much more powerful than a function of a classical programming language.

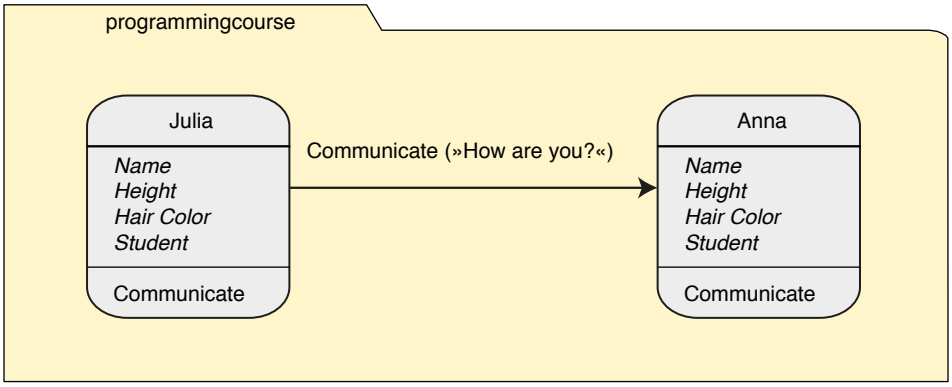


Figure 3.6: Objects communicate by calling methods.

But no matter what the term is called, one thing is the same: behaviors like *communicating* determine an object's ability to understand and perform tasks. So objects can be controlled by methods. Not only one type of method exists, but there are the following five basic types: *Constructors* (»Builders«), *Destructors* (»Destroyers«), *Modification Methods* (»Setters«), *Query Methods* (»Getters»), and *Operations* (»Functions«).

Constructors

As in natural life, objects of a computer program also have a life cycle. They are born, that is, created, and at some point they die, that is, they are deleted and their memory is released. The most important method is the one that creates an object. Consequently, they are also called »constructors«. They construct, that is, create an object.

Destructors

Methods that destroy an object are called »destructors« in object-oriented programming. In some programming languages, you can call these destructors directly and thus destroy an object immediately. In Java this does not work. Here, an object is automatically destroyed when it is no longer needed. More about this later.

Setter Methods

Methods that change the value of an attribute are called »setter methods« or »setters« for short. They change the state of the object. Such a method can be used to change the speed at which *Anna* moves (Figure 3.7). The corresponding method is called *running* and has a so-called parameter that specifies the new state, the *speed*.

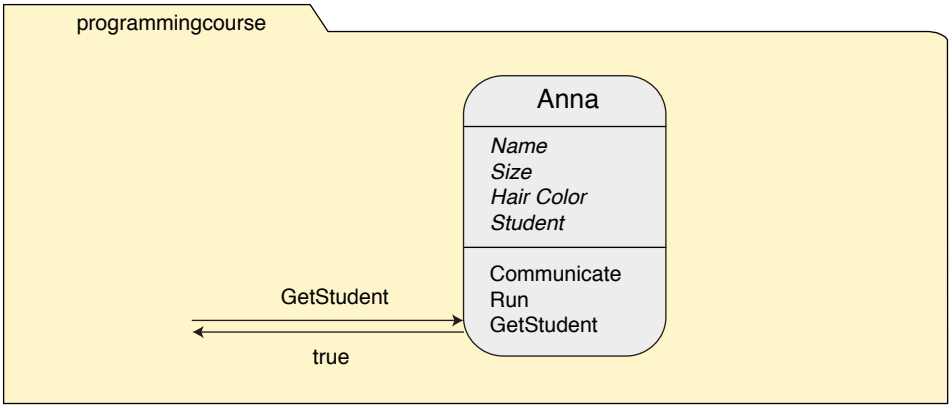


Figure 3.7: The getter method »Run« changes the state of »Anna«.

Getter Methods

Getter methods, or »getters« for short, are methods that only query a specific attribute. They do not change the state of the object. For example, one such method would be to query whether Anna is enrolled at a university (Figure 3.8). This method has a so-called return value: the status of the student, which can be answered with *true* or *false*.

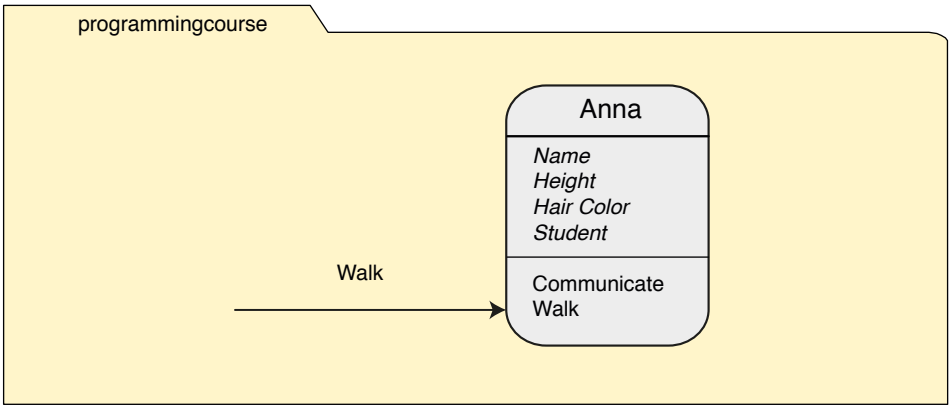


Figure 3.8: The getter method inquires about Anna's student status.

Operations

Methods, which accomplish for example only one arithmetic operation, are called in object-oriented programming mostly operations or functions. Nevertheless, they must not be confused with the functions of the same name in classical programming languages, because like other methods they are inherited from class to class (Section 3.6, »Inheritance«). In addition, they can be overloaded and overwritten (Section 3.13, »Polymorphism«).

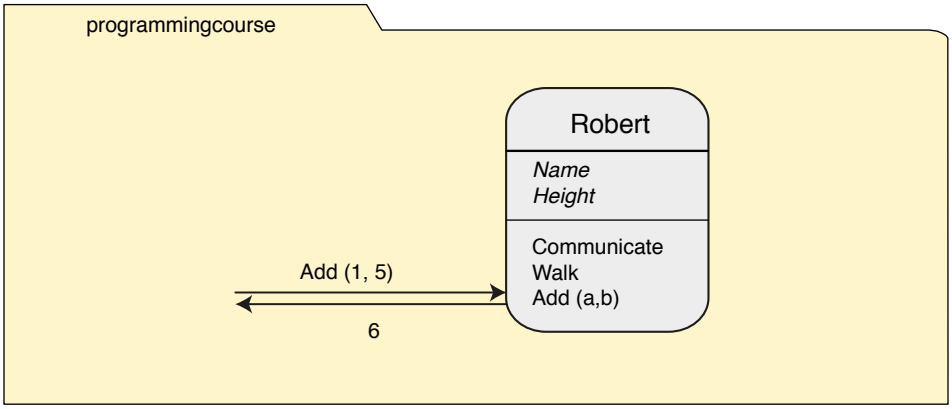


Figure 3.9: The operation »Add« returns the sum as the result.

3.5 Abstraction

Perhaps you will now say, »That's all nonsense. A person's abilities and attributes are much more complex and cannot be reduced to size and color and understanding.« This is true in the natural world, but it is usually completely wrong in the artificial world of software development.

It would be correct only if one had to fully represent nature in a program. But for such an exaggerated accuracy there is rarely a reason in programming. Object-oriented programming makes it easier to represent the real world as naturally as possible and thus promotes good software design.

Figure 3.10: Abstraction gives you the essence of a class.

It seduces thereby also to exaggerated constructions. The art is to counteract this and to represent reality as accurately as necessary, but as simply as possible. As you will see later with larger example programs, just the analysis of the components essential and correct for the program often causes great problems.

If you program within a program only the parts essential for the functionality, then this has practical reasons: The program can be developed faster, it becomes cheaper and leaner. Thus, it requires less memory, and it will usually run faster than a program that is overloaded with unnecessary information.

To achieve this compactness, it is necessary to abstract, i.e. simplify, the usually extremely complicated natural objects and their relationships as much as possible. The technical term for this technique is therefore also called *abstraction* (Figure 3.10).

3.6 Inheritance

After the introduction of classes, objects, methods and attributes, it is time to relate these new concepts to the notion of *inheritance*. Inheritance allows behavior to be transferred between classes, and therefore between objects, using a blueprint. Inheritance copies attributes and methods of the base class to the derived class.

To clarify this, again an example: Humans and robots are in some respects similar, but in various respects nevertheless extremely different. These differences are of a different order than the differences between two humans: Humans and robots have a distinctly different shape (Figure 3.11).

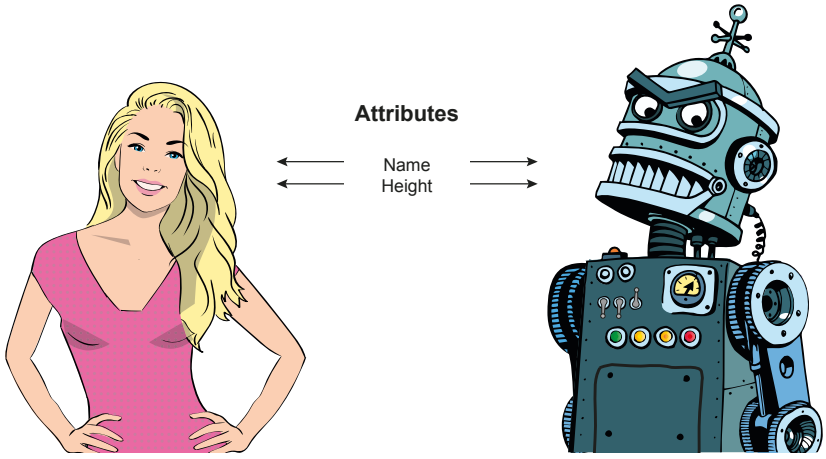


Figure 3.11: Objects of different classes differ in their shape.

The fact that people have a different shape than robots in the sense of the theory of forms does not really need to be emphasized in view of Robert's bulky appearance. To get to the heart of the differences, it helps if you simply try to reconcile the attributes of people listed on the previous pages with those of a robot. As you will see, this only works for a certain subset of the attributes.

What does this mean for programming? It means that you can find out whether objects belong to a common class in the way just shown. In all cases where commonality matters,

you should assign a common class to the objects. In all cases where the differences between objects matter, it is better to assign them separate classes.

3.6.1 Base Classes

So far, we have emphasized the differences between humans and robots. What happens if we look at it from the other side: What do a human like Anna and a humanoid machine like Robert have in common? From a programming perspective, every human like Anna and every robot like Robert have a certain size. They can both walk and communicate.

They share the aforementioned characteristics with a variety of living beings. Are robots living beings? Probably not. Most people would certainly deny them the ability to live. However, they could be called artificial beings or machine beings. From the point of view of object-oriented programming, this means that we could assign humans and robots to a common base class *Creature*. Figure 3.12 shows what such a base class would look like.

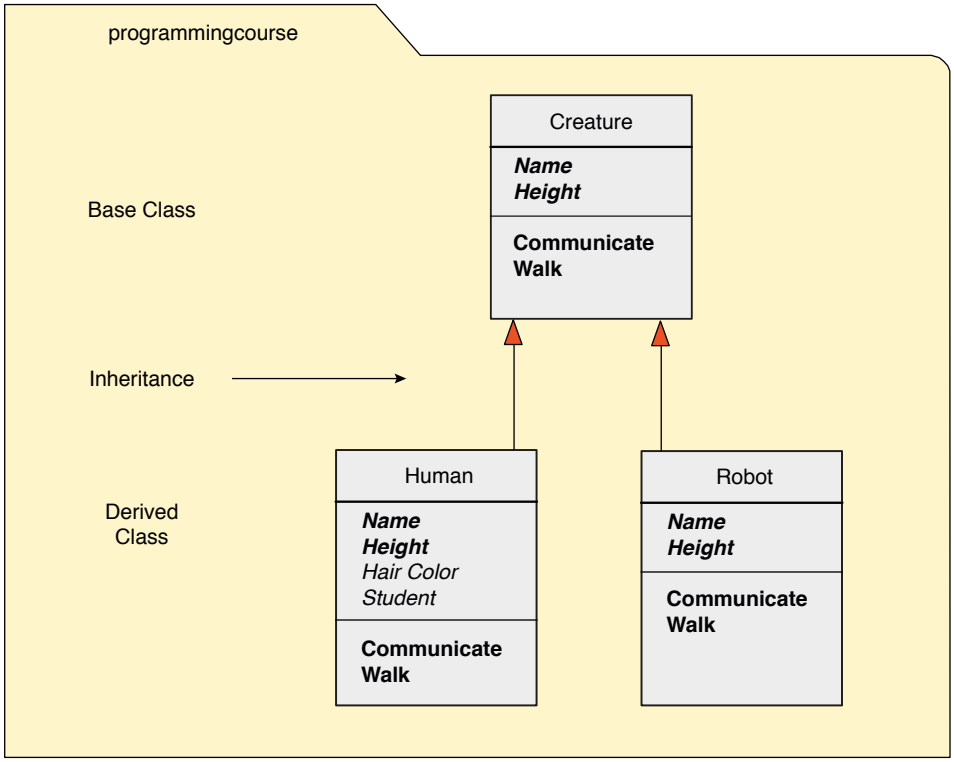


Figure 3.12: The base class transfers base properties and behavior.

Note especially the words in bold: To avoid having to reprogram the attributes *Name* and *Size* as well as the behavior *Communicate* and *Walk* for each of the classes *Human* and *Robot*, it makes sense to move this behavior to a base class. The bold attributes and methods are the *common features* of both classes from the point of view of object-oriented pro-

gramming. The common attributes of the base class not only make programming easier. They also unify all derived classes.

3.6.2 Derived Classes

Suppose you want to create a new class called *Animal* based on the class *Creature*. In object-oriented programming, you would say you derive a new class called *Animal* from the class *Creature*. As in genealogy with family trees, one also says the class *Animal* derives from the class *Creature*. The new class *Animal* would inherit the attributes *Name*, *Height* and the behavior with *Communicate* and *Walk* from the base class *Creature*, as did the classes *Human* and *Robot* before. Attributes and behavior thus inherit (Figure 3.13).

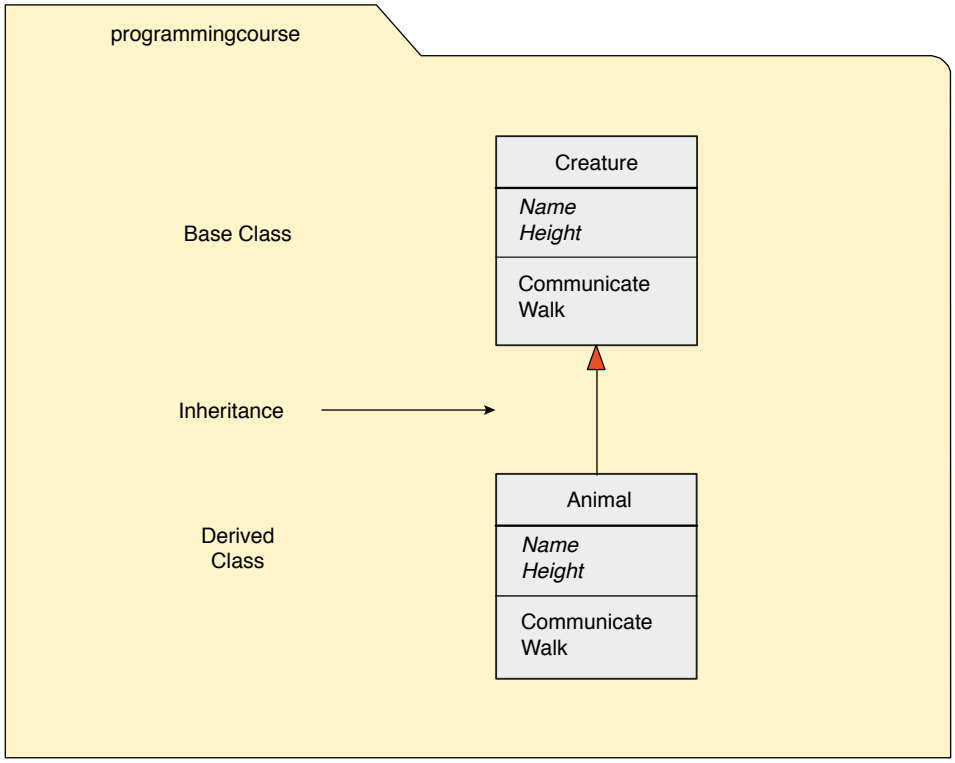


Figure 3.13: The new class »Animal« is a derived class from »Creature«.

3.6.3 Multiple Inheritance

It is common in nature, but not allowed in the Java programming language for good reason: multiple inheritance. It would come in handy if you wanted to merge two classes, for example the class *Human* with the class *Horse*. The new hybrid *Centaur* would inherit attributes

and behavior of both base classes (Figure 3.14). But what attributes and behavior? Should centaurs communicate and walk like humans or like horses? Can centaurs be students?

As nice as the example in mythology is, with such scenarios software development comes to the limit of what is technically reasonable. It just does not make sense to transfer hereditary information at random in order to imitate nature. As a rule, the user wants programs that have defined properties and whose behavior is predictable. For the reasons mentioned, the developers of the Java programming language have deliberately decided against conventional multiple inheritance.

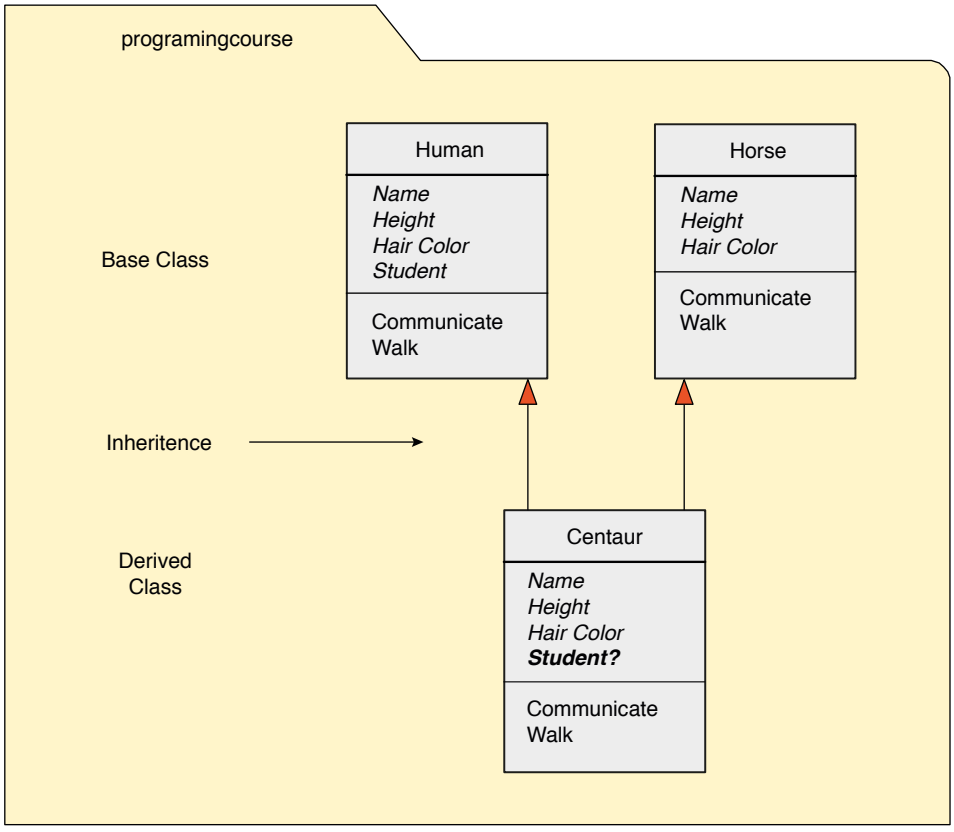


Figure 3.14 Multiple inheritance using the example of a hybrid

How you can still connect multiple base classes without side effects is presented in chapter 9, "Classes and Objects", section 9.5, »Interfaces«.

3.7 Access Protection

One of the most important features of object-oriented languages is the protection of objects and their attributes from unwanted access. You may remember the beginning of

this chapter. Object-oriented programming was invented to overcome the software crisis caused by buggy software. The software should become more robust by the new programming.

For the reason, each object has a kind of capsule that protects the data and methods of the object. The capsule hides the parts of the object that should not be accessible from the outside or only by certain other objects. The places where the capsule is permeable are called *interfaces*. The idea of the whole thing is to let through to an object only the information that absolutely must reach it. Therefore, each object can be designed like a cell (Figure 3.15).

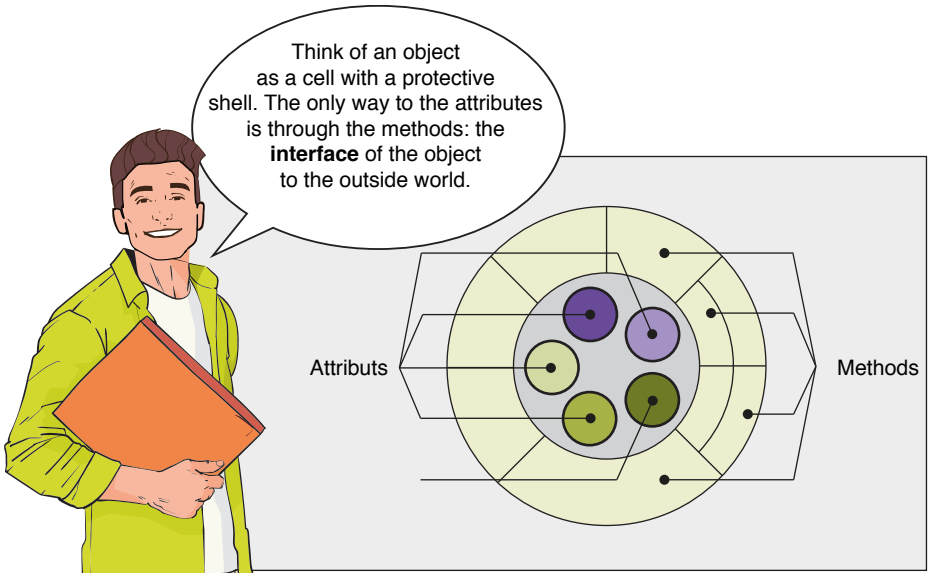


Figure 3.15: Like a cell, the object protects its attributes from unwanted access.

The most important interface of a class is its constructor. Let's take the class *Robot* as an example. The object *Robert* can be created via the constructor of this class. Another example of such an interface is the method *Understand* of the class *Robot*. It is publicly accessible. In contrast, its internal method *Add* is not publicly accessible. Anna communicates with *Robert* through this interface and uses it to tell *Robert* what to compute (Figure 3.16).

All parts that may be accessed from the outside are marked with a plus sign. All parts that cannot be manipulated from the outside are marked with a minus sign. However, the object *Anna* is not allowed to change all data of *Robert* via this interface. For example, Anna should not be allowed to change the name of the robot at all. If there were a publicly available method such as *Rename*, she could use it to change *Robert*. But this is not allowed like this.

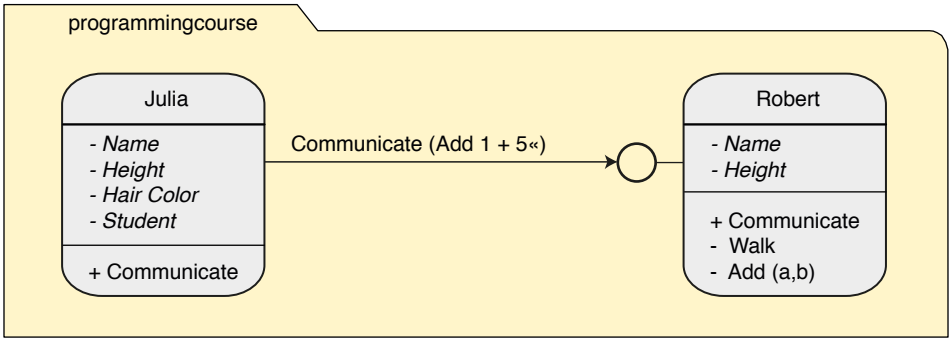


Figure 3.16 Objects communicate only through their interfaces.

3.8 Relationships

Classes and their objects maintain a wide variety of relationships with each other in a program. In the previous sections you have already learned about several forms of relationships. Basically, there are relationships without inheritance and relationships with inheritance.

3.8.1 Relationships Without Inheritance

Object-oriented programming is very strict about relationships. It knows three different types of relationships without inheritance (Figure 3.17).

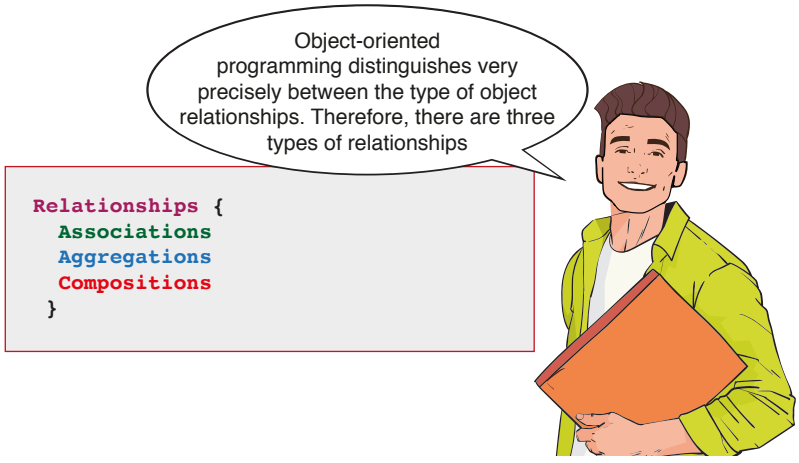


Figure 3.17: Away from the inheritance relationship, there are three relationship types.

3.8.1.1 Association

Association is the simplest form of a relationship between classes and objects. The dependencies are small in this type of relationship compared to inheritance. The objects are also said to be loosely coupled.

For example, an association exists when an object named *Anna* sends a message *Robert* to an object named *Add* (Figure 3.18). The two objects *Anna* and *Robert* exist separately and do not inherit anything from each other.

Figure 3.18: A simple association between humans and robots

3.8.1.2 Aggregation

An increase of the association is the aggregation. Such a relationship exists when an object is composed of other objects. For example, a robot is said to consist of an unspecified number of screws (Figure 3.19). This means, for example, that a robot maintains a »consists-of« relationship (part/whole) with the screw.

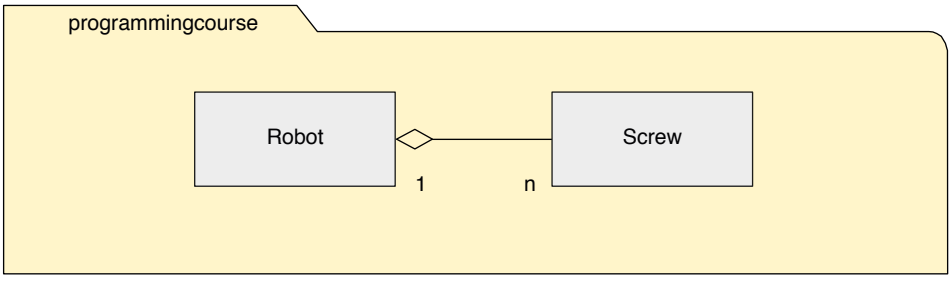


Figure 3.19: Aggregation between robot and screw

However, this relationship is of a completely different quality than in the previous example between a human being and a robot. While man and robot can exist alone and independently of each other, the robot is composed (among other things) of screws. Again, it is important to note that both objects do not inherit anything from each other and each screw object is also viable on its own. This example differs from the stricter composition.

3.8.1.3 Composition

The strongest form of relations, which are not based on inheritance, is the *composition*. As with aggregation, there is again an »consists-from« relationship, but it is again stronger in contrast to aggregation. The dependencies are again stronger.

An example of a composition is the relationship between a human being and his two legs. Here, fortunately, there is a very close relationship, because a leg – in contrast to a screw – is completely meaningless as an independent object. When a human object is created, it automatically gets two individual legs, which cannot be used in connection with other objects.

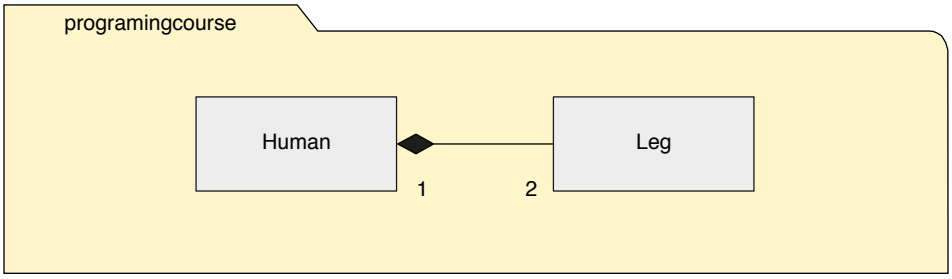


Figure 3.20: A human and his two legs as a composition.

Thus, human legs are not viable without a suitable object of the class *Human*. When a human object dies, so do its human legs.

3.8.2 Inheritance Relationships

Inheritance relations are also called generalization or specialization. These are not subtypes of inheritance, but alternative terms for inheritance relations. Which of the two alternative terms you want to use depends on the point of view from which you look at the inheritance relation.

3.8.2.1 Generalization

If you want to look at the base class from the point of view of the derived class, generalization is the appropriate term for it. For example, the class *being* is a generalization of the classes *Human* and *Robot*. In other words, class *Creature* is the general term (= generalization) for classes *Human* and *Robot*.

3.8.2.2 Specialization

If you want to look at the derived class from the point of view of the base class, specialization is the appropriate term for it. For example, the classes *Human* and *Robot* are a specialization of the class *Creature*. In other words, the classes *Human* and *robot* are a refinement of the class *Creature*.

3.8.2.3 Inheritance Can Be Problematic

Inheritance relationships establish a very strong coupling between classes and thus also between objects. Such a strong coupling has not only advantages, but also serious disadvantages, as the following example shows:

A class named *Fish* is to be created from the class *Animal*, which in turn descends from *Creature* (Figure 3.21). The new class inherits the *name* and *size* attributes, as well as the *Communicate* and *Walk* methods. Wait a minute: *communicate* and *walk*? This is where you start to wonder. Can fish communicate? Maybe. But walking, with a few exceptions, fish certainly can't.

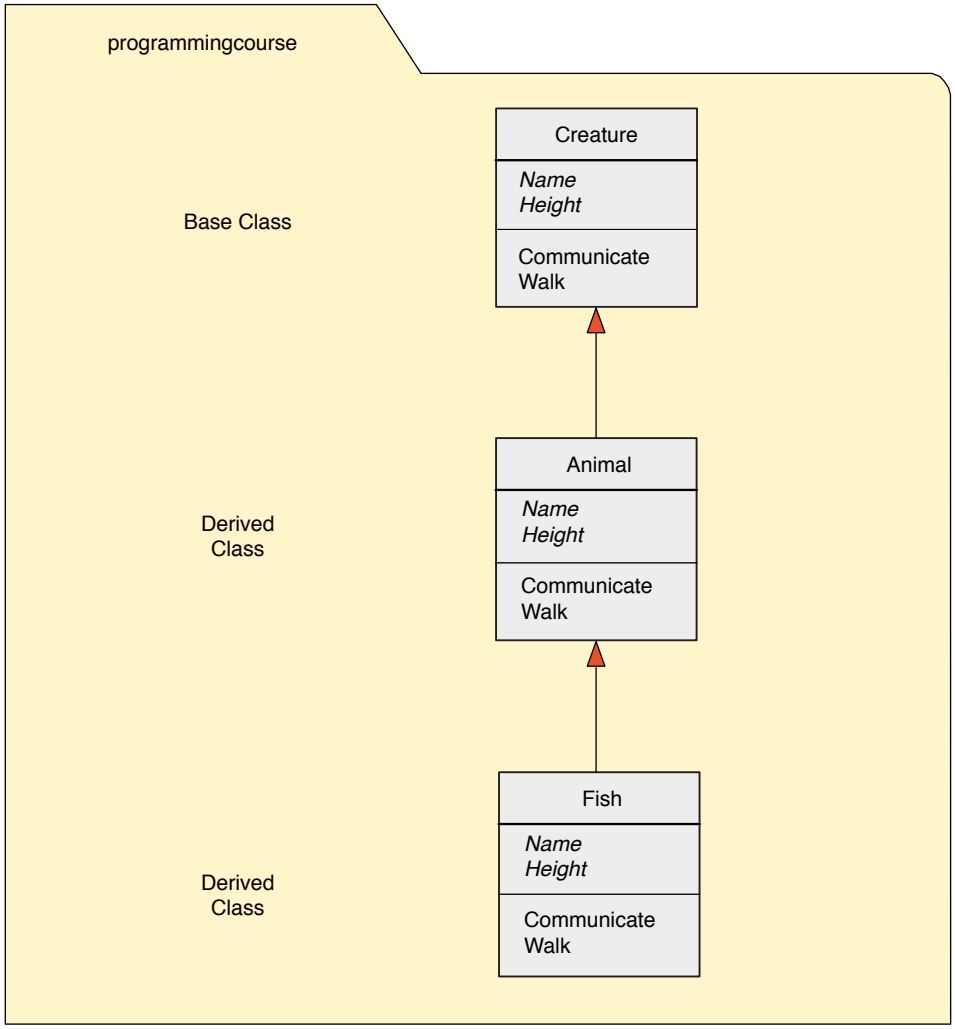


Figure 3.21: Design flaws are also passed on through inheritance.

What has happened here is exactly what is one of the daily problems of object-oriented programming: The functionality of the base class has not been sufficiently analyzed. To

put it simply: There is a design error here, which has to be fixed by replacing at least the method *Running* with the method *Moving*. But this would have some consequences for both classes *Human* and *Robot*.

3.9 Design Flaws

You can perhaps imagine that it is very inconvenient when the base class has to be changed due to a design error. Due to the strong coupling between the base class and the derived class, any changes propagate like an avalanche to all parts of the program where objects of type *human* and *robot* with method *running* were used. In all places of the program where the *running* method of the *being* class was used, it must be replaced by the *moving* method (Figure 3.22).

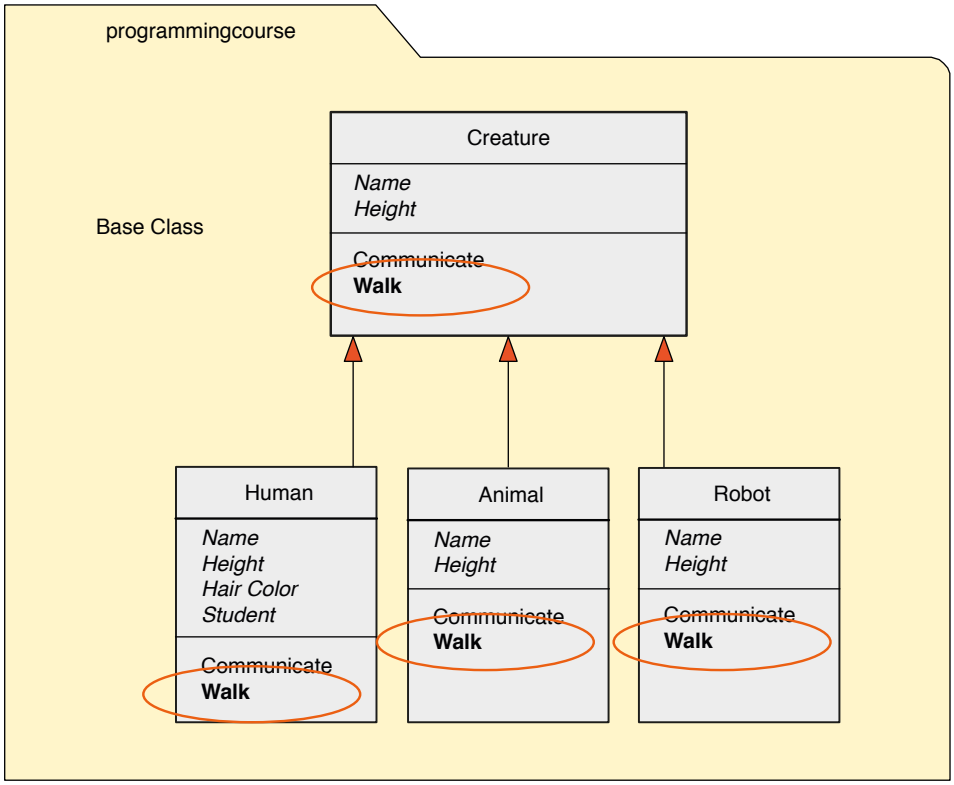


Figure 3.22: Refactoring can fix inheritance errors.

In the case of design errors, the technique of inheritance turns out to be a major drawback. In addition to this shortcoming, inheritance also has the disadvantage that not only design errors but all other excellently designed but undesirable parts of the base class are transferred to the derived classes in the form of ballast: the descendants of such overweight

classes become fatter and fatter. Therefore, you should always look at inheritance critically, use it sparingly, and really only use it where it makes sense.

3.10 Refactoring

But back to the design errors. How do you deal with errors of this kind? They are not as big a problem today as they were a few years ago, despite inheritance. With a tool like Eclipse, it is relatively easy to do the necessary restructuring (refactoring). However, if possible, you should only restructure software during the analysis and design phase of the software. As a rule, the later changes are made, the higher the associated effort. Chapter 20, »Development Processes«, highlights the topic again in more detail.

3.11 Modeling

To avoid such design errors and thus costly restructuring, it makes sense to design a model of the software for larger projects. Just as in automotive engineering one develops a model before each new automobile to be constructed, it also makes sense in software development to construct a model before starting the actual implementation of the project. A model that represents a faithful reproduction of a complete section of the software is called a prototype (sample, precursor).

3.12 Persistence

A program creates objects that are destroyed again at the end of their life. These objects are called transient. Sometimes, however, a »life after death« is also desirable for objects. They should also be brought back to life when the program is finished and the user of the program goes home. The next day the user starts the program again and wants to continue working with the same object.

Such »immortal« objects are called persistent. This means nothing more than that they are stored in a suitable form. They are then in a kind of deep sleep in a file on a hard disk or in association with other objects in a database.

3.13 Polymorphism

The name polymorphism comes from the Greek and means as much as multiformity, diversity. The term sounds more like mineral science than computer science, and so you may

not be surprised that the chemist Mitscherlich discovered polymorphism in minerals at the beginning of the 19th century. He noticed that some minerals, such as calcium carbonate ($CaCO_3$), can take on different crystal forms without changing their chemical composition. That is, they can take on a different shape depending on pressure and temperature.

All very nice so far, but what does this have to do with object-oriented programming? It certainly doesn't mean that an object like *Robert* can change shape as radically as a mineral. It means that *Robert* can react differently depending on the situation if »programmed« skillfully. Sounds like magic, but it is not.

3.13.1 Static Polymorphism

Imagine that the object *Anna* tells the object *Robert* that *Robert* should perform an addition with two integer values. What will happen? - Of course, it is no problem for *Robert* to add these values and tell *Anna* the result. But what would happen if *Anna* again tells *Robert* to add, using fractions? Either *Robert* would decompose the task or it would perform addition directly with fractions, because it would have an internal method for doing so (Figure 3.23).

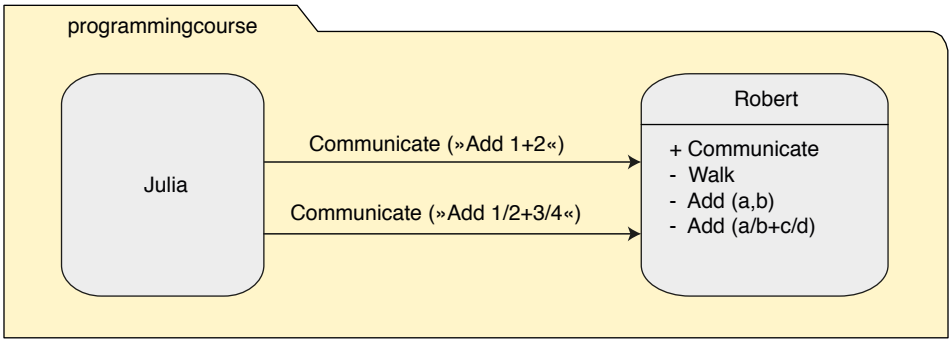


Figure 3.23: »Robert« has two differently designed methods called »Add«.

In order for *Robert* to follow the various instructions of *Anna*, it needs methods of »different shapes«. It needs a method that responds to two parameters speed and a method that responds to three parameters. Although the methods have the same name, they result in different processing by the object *Robert*. The technical term for this technique is *Overload*.

3.13.2 Dynamic Polymorphism

The situation is different from multiple inheritance if you want to deliberately bypass properties of the base class during inheritance. For this purpose, I would like to go back to the example of the base class *Creature*. Suppose you want to determine in the derived class *Robot* how robot objects communicate. To do this, you *override* the *Communicate* method and specify the way of understanding in the *Robot* class for the derived objects.

Method overriding is a very powerful tool of object-oriented programming. It allows you to partially or completely suppress unwanted inherited information and thus compensate

for possible design errors – within limits – or fill gaps in the base class. The technique is extremely simple. It is enough to describe an identical method in the derived class *Robot*, so that objects like *Robert* »suddenly« behave differently.

3.14 Design Rules

Object-oriented programs are no guarantee for cleanly structured and logically built programs. Object-oriented programming facilitates good software design, but it does not force it. Therefore you should consider some basic rules. They make it easier for you to develop lean, fast and robust Java programs (Figure 3.24).

```
Design Rules {
• Avoid inheritance
• Reduce the requirements
  to the essentials
• Protect all internal
  attributes and methods
• Draw a domain-oriented model
  with all dependencies
• Work with a prototype
}
```

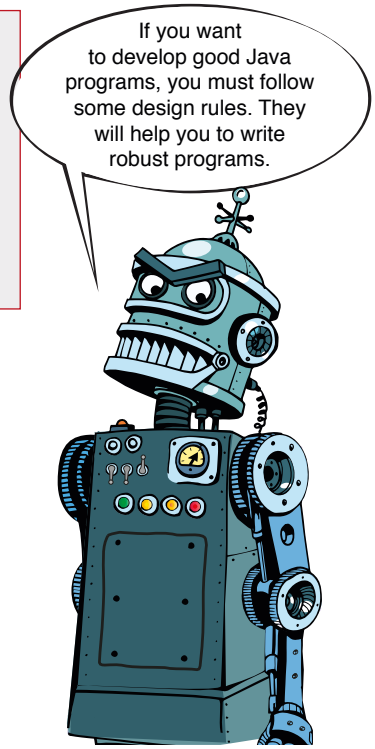


Figure 3.24: Robert recommends some basic rules for good design.

3.15 Summary

Object-oriented programming was a response to the software crisis in the mid-1960s. Object orientation makes it easier to translate the natural world into computer programs. These object-oriented computer programs consist of one or more objects.

The objects have their own memory. Classes model the common behavior of their objects. Each object is a copy of its class. Objects communicate with each other via message exchange. A program is executed by giving control to the first object and treating the rest as its message.

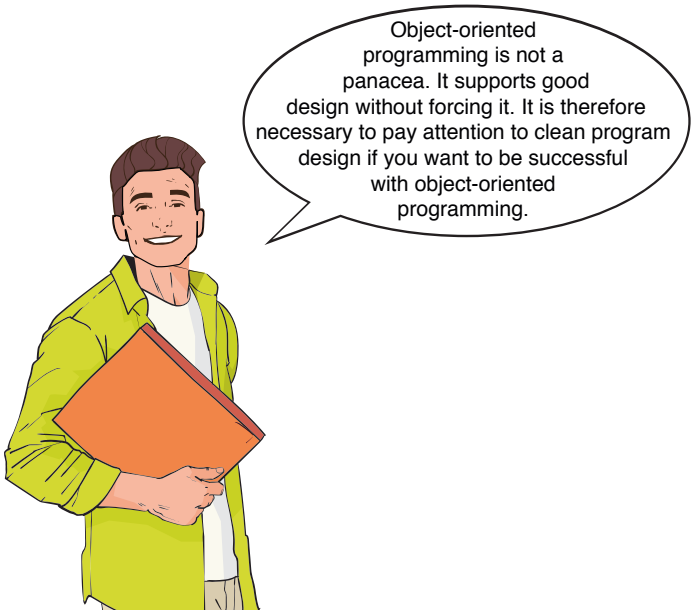


Figure 3.25: Florian summarizes the specifics of object-oriented programming.

An object can be compared with a natural living creature and has a shape and abilities. The shape is determined by attributes, while the capabilities are determined by methods. Both components of an object are defined in the class from which an object originates. It provides the blueprint for similar objects.

3.16 Literature

Wikipedia: Alan Kay; *https://en.wikipedia.org/wiki/Alan_Kay*

3.17 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/object-oriented_programming
There you will also find the solutions to the exercises.

4 Development Environment

4.1 Introduction

The development environment you need for this book consists of a variety of tools. Before you can really get started and develop your first Java program, you must carefully install these tools. This chapter explains how to do that in the first section. In the second section, the chapter gives you an overview of the tools' functions.

Figure 4.1: Julia guides you through this chapter.

Installations are always a nuisance. But even more annoying are faulty installations. They spoil the fun by making hard-to-understand mistakes when developing Java programs. You can avoid these errors by reading this chapter even if you should already have Java and/or an Eclipse development environment installed on your computer.

4.2 Installation

4.2.1 Operating System

You need a computer with a 64-bit operating system as the basis of the software installations. As operating systems you can use, for example, Windows 8.1 or from 10, Ubuntu Linux from 18 or 19 and MacOS from 10.13.

Determine System Type Under Windows

In Windows 8.1 and Windows 10, you determine the system type via the Control Panel. To do this, simply click START and then SETTINGS → SYSTEM → INFO. Under SYSTEM TYPE you will then see whether the operating system is 32-bit or 64-bit (Figure 4.2).

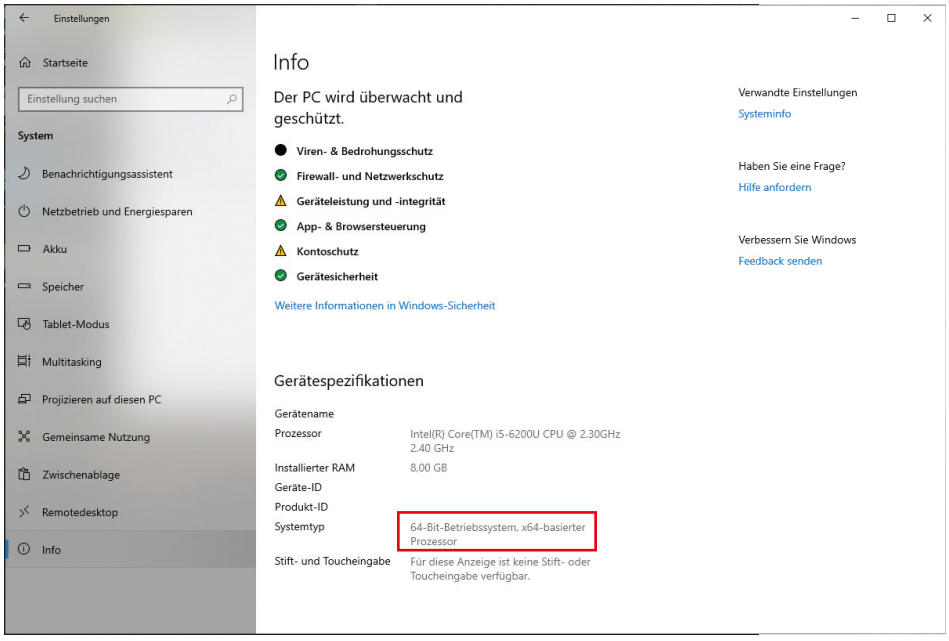


Figure 4.2: The software of this book requires a 64-bit Windows.

Determine System Type Under Linux And MacOS

Under a Unix operating system such as Linux or MacOS, the quickest way to determine which system type you are working with is to use a Unix command. To do this, start a terminal via the graphical interface and then enter the following command:

```
uname -m
```

Should *x86_64* be displayed here, everything is fine. In this case it is a computer with a 64-bit processor and a 64-bit operating system, which is necessary as a basis for the installation on the following pages.

4.2.2 Install Java

The Java programming platform consists of Java tools and the Java Runtime Environment (JRE). Together, the combination of both is called Java Development Kit or JDK for short (Chapter 2, »Technology Overview«).

You should have at least version 13 of the Java Development Kit installed on your computer for this book. JDKs are available from some Internet sites in both 32-bit and 64-bit versions. Be sure to download a 64-bit version, since the "Eclipse" version that this book requires will only work with such a runtime environment.

Figure 4.3: Auf der Oracle-Seite erhalten Sie ein JDK mit Installationsprogramm.

You can get the JDK for all operating systems from the download page of the Java owner Oracle *https://www.oracle.com/technetwork/java/javase/downloads/index.html* (Figure 4.3) or from the OpenJDK download page *https://jdk.java.net*. The JDKs on the Oracle site come in two different packages, one of which includes an installer to facilitate installation.

Install Java for Windows

On Windows, first determine whether Java is already installed and in which version. To do this, first start a command prompt via the Start menu and then enter the command *java -version* (Figure 4.4).

If the error message »The command 'java' is either misspelled or could not be found« appears or a Java"=version before Java 13 is displayed, install a newer Java version. The easiest way to install is to use the installation program available from the Oracle site

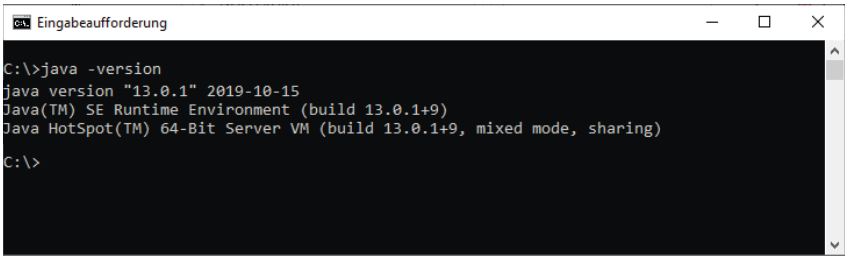


Figure 4.4: Use the command prompt to determine which Java version is used.

mentioned at the beginning of this section. Start this program and follow the installation instructions. If you decide to use OpenJDK, download the zip file and unpack it into a directory of your choice, for example into the directory *C:\programs\OpenJDK13* or *C:\programs\Java\OpenJDK13*.

For both the Oracle and OpenJDK versions, you should then check the search path and adjust it if necessary. To do this, open the Control Panel. On Windows 10, type ENVIRONMENT VARIABLES in the search box in the upper right corner and then click the EDIT ENVIRONMENT VARIABLES FOR THIS ACCOUNT link. The ENVIRONMENT VARIABLES dialog that you will see is split into two parts. The upper pane shows USER VARIABLES and the lower pane shows SYSTEM VARIABLES. Change the path of the system variables and enter the path for the bin directory of the JDK there (Figure 4.5), if the installation program has not already done so.

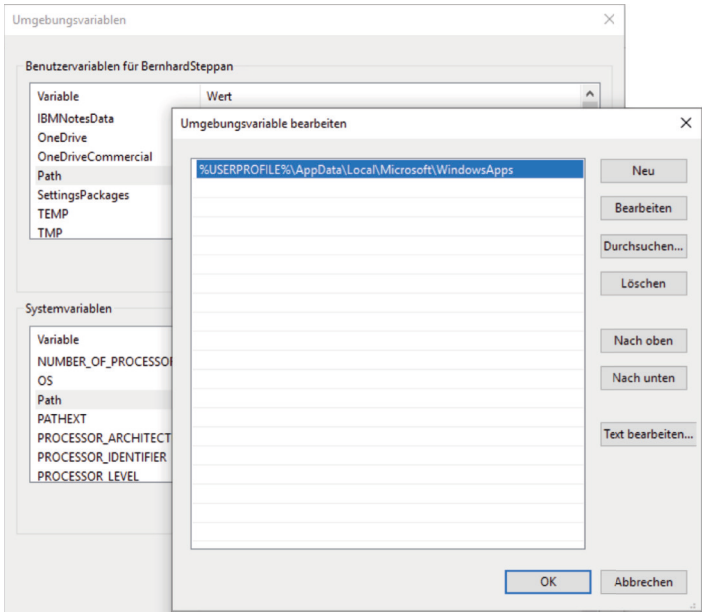


Figure 4.5: Here you enter the path to the JDK directory.

Once the path has been adjusted, check again using the command prompt by entering the command *java -version* to see if the operating system has correctly detected the new JDK version and search path. The version of the JDK you just installed should now appear. If this is the case, you can scroll forward to the 4.2.3 section, "Install Eclipse", and install Eclipse.

Install Java for Linux

In the following I describe the installation for the Linux derivative »Ubuntu«. For most other Linux distributions, the procedure works identically. It is important that you first check whether there is already a correctly installed Java version on your Linux computer. You can do this by opening a terminal and entering the following command:

```
java -version
```

If the Linux terminal responds with an error message like »Command 'java' not found, but can be installed with ...« then Java is not installed. If a version is already displayed that is not at least version 13, install a current JDK. This works as follows with a terminal:

```
sudo apt install default-jre
```

After that, the *java -version* command should succeed and display the Java version you just installed. If this is the case, go to the 4.2.3 section, "Install Eclipse", and install the Eclipse"=development environment.

Install Java for MacOS

On MacOS, first check if there is already a newer Java version on your Mac. The easiest way to do this is with the MacOS Terminal. It is located under the Utilities of the Applications folder. Start the terminal and type *java -version*. If the error message »No Java runtime present, requesting install.« appears, the operating system automatically pops up a dialog where you can click the MORE INFO button to automatically go to the download page for the latest Java version (Figure 4.6).

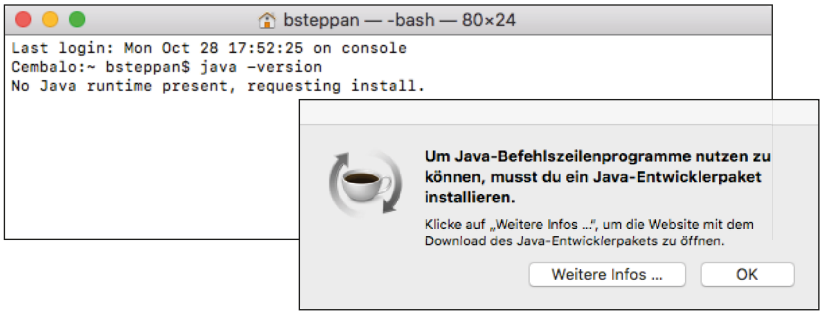


Figure 4.6: The terminal provides details about if and what version of Java is installed.

If the terminal displays an older Java version before Java 13, install a current Java version from one of the websites mentioned at the beginning. If you choose the »Oracle« site, select a DMG file. The DMG file creates a temporary drive that contains an installer. This installer will guide you through the installation.

With the OpenJDK version the installation is not quite so simple, because here only a compressed file is available. Download this compressed tar file and unpack it in any folder like *jdk-13.0.1.jdk*. Then move the complete folder *jdk-13.0.1.jdk* into the directory */Library/Java/JavaVirtualMachines*. In this directory the MacOS searches for Java versions. You can get to the directory by using the Finder menu GO TO and the menu command GO TO FOLDER and entering the directory path (Figure 4.7).

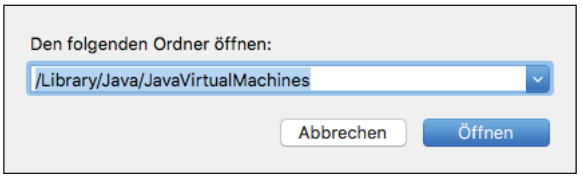


Figure 4.7: The Finder will take you to the installation directory.

After this manual installation, you must restart the computer in any case so that the operating system recognizes the new Java version. After rebooting, check again by typing *java -version* in the terminal to see if the operating system displays the correct JDK. If this is the case, you can go to the next section to install the Eclipse development environment.

4.2.3 Install Eclipse

Eclipse Versions

The Eclipse development environment is not only a development environment for Java. It has also been developed on the basis of Java itself. For this reason, you need a suitable Java Runtime Environment to run Eclipse as you would for any other Java program. Since the newer versions of Eclipse *only* are available as 64-bit versions, it was necessary to install a 64-bit JDK.

The versions of Eclipse have names like »2019-12«. The name »2019-12« means that this Eclipse development environment was released in December 2019. In addition to these version names, there is also a version number that is incremented with each new Eclipse"=version. Note that only the newer Eclipse versions as of »2019-12« are fully compatible with Java 13 (table 4.1).

Table 4.1: Only newer Eclipse versions are fully compatible with Java 13.

Name	Release Date	Version	Java Support
Eclipse IDE 2019-06	June 2019	4.12	Java 12
Eclipse IDE 2019-09	September 2019	4.13	Java 12 (Java 13 with extension)
Eclipse IDE 2019-12	December 2019	4.14	Java 13
Eclipse IDE 2020-03	March 2020	4.15	At least Java 13

Eclipse Packages

Eclipse is more universally oriented than any other development environment. It is available for the most diverse programming languages and the most diverse areas of application

in software development. For this purpose, the Eclipse"=development environment can be extended by so-called plug-ins.

In order to facilitate the adaptation to the most diverse areas of application, there are preconfigured packages for the most diverse target groups. These packages save you from having to install and configure many plug-ins yourself. The table 4.2 shows the most important preconfigured packages for Java developers.

Table 4.2: The most important Eclipse packages for Java developers.

Description	Target Group
Eclipse IDE for Enterprise Java Developers	Java Enterprise Developers
Eclipse IDE for Java Developers	**Eclipse Beginners**
Eclipse IDE for Web and JavaScript Developers	Web Developers
Eclipse IDE for RCP and RAP Developers	RCP and RAP Developer [1]

For this book, you only need the package »Eclipse IDE for Java Developers«. It covers approximately the Java Standard Edition (JSE). If you are already using an older Eclipse version, install this Eclipse version additionally. This ensures that you can easily follow the illustrations and descriptions in this book.

Figure 4.8: Only the package »Eclipse IDE for Java Developers« is required for this book.

By the way, the abbreviation »IDE« stands for »Integrated Development Environment«. The term »SDK«, which means »Software Development Kit«, is also frequently used in the literature in connection with Eclipse. This means that Eclipse is a collection of tools for software development.

Download

For Eclipse, an installer has been available for some time to put together an Eclipse package yourself. This is very practical for experienced users, because they can configure their development environment themselves. To get started with Eclipse, I recommend downloading a pre-built package of the development environment instead. To install such a package, you only need to unpack a compressed archive file.

To download this file from the Eclipse homepage, go to the web page with the address *https://www.eclipse.org/downloads/packages*. There you will find all the Eclipse packages previously mentioned in table 4.2. Then select the appropriate 64-bit package »Eclipse IDE for Java Developers« for your operating system and download the archive file to your computer.

Install Eclipse on Windows

On Windows, create a directory named *Eclipse* in the Windows program directory. You will need administrator rights for this. Then unpack the archive file you just downloaded and rename the *Eclipse* directory it contains to *Eclipse-2019-12-JSE*. After that, copy this directory completely into the just created directory named *Eclipse* (Figure 4.9).

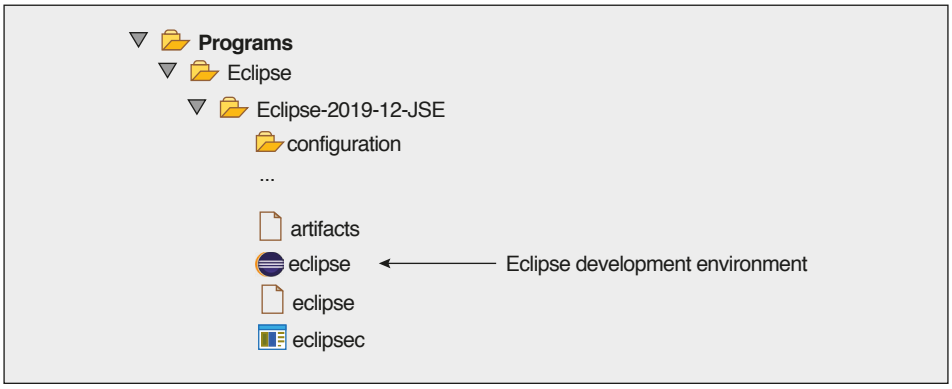


Figure 4.9: The installation directory under Windows

Finally, you can assign a unique alias such as *Eclipse-2019-12-JSE* for the Eclipse development environment in the edition for the Java Standard Edition (JSE). After that, copy this alias to the start menu or to the desktop so that you do not have to search for the installation directory every time you start the Eclipse development environment. This completes the basic installation of Eclipse. You can now scroll forward to the 4.2.3, »Configure Java Version« section to configure the Java version for Eclipse.

Install Eclipse on Linux

On Linux, create a subdirectory named *eclipse* in the *opt* directory. You need administration rights for this. Then unpack the tar archive and rename the Eclipse directory inside it to *eclipse-2019-12-jse*. Next, move the entire *eclipse-2019-12-jse* directory to the directory you just created named *eclipse*. The directory should then look like figure 4.10.

Finally, you can assign a unique alias named *Eclipse-2019-12-JSE* for the Eclipse development environment in the edition for the Java Standard Edition (JSE). Then copy this alias to the desktop or to the program menu. This has the advantage that you have Eclipse immediately at hand. It also makes it easier to distinguish the Eclipse version you have just installed from others that you may install later. This completes the basic installation. You can now advance to the 4.2.3, »Configure Java Version« section to configure Eclipse for the installed Java version.

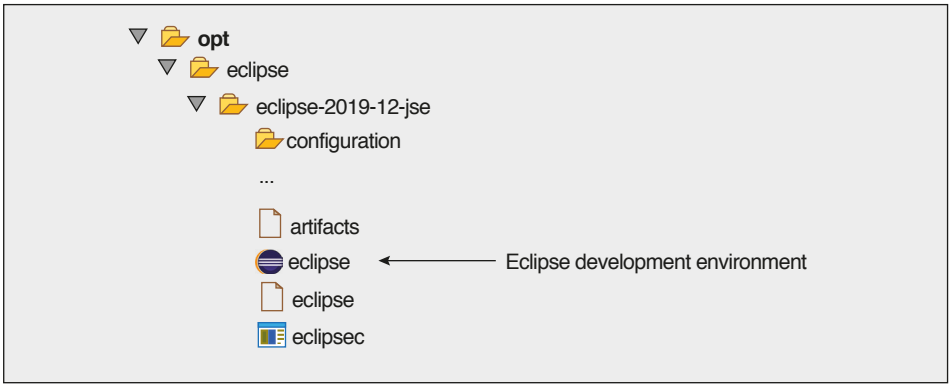


Figure 4.10: The installation directory under Linux.

Install Eclipse on MacOS

Create a directory named *Eclipse* with a subdirectory *Eclipse-2019-12-JSE* in the Applications folder of MacOS. You will need administrator privileges to do this. Double-click the downloaded DMG file. MacOS creates a temporary drive from the DMG file that contains the Eclipse development environment. Copy the *Eclipse.app* application into the *Eclipse-2019-12-JSE* folder you just created. After that, the folder should look like in figure 4.11.

To finish the Eclipse installation, you can still assign a unique alias called *Eclipse-2019-12-JSE* for the Eclipse"=development environment in the edition for the Java Standard Edition (JSE). Copy the alias to the Dock or the desktop - not the program icon, otherwise the Finder will only display the name *Eclipse*. The alias in the Dock has the advantage that you can better distinguish the currently installed Eclipse version from others that you might install later. Once this is done, you can go to the next section and configure the Java version for Eclipse.

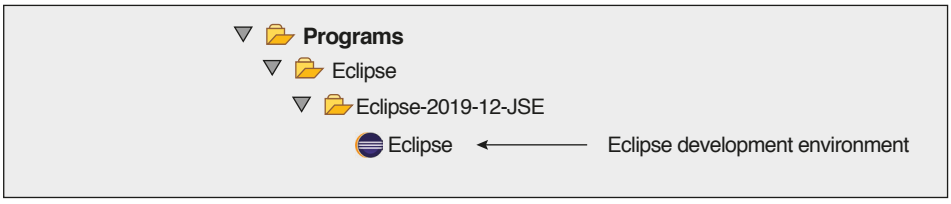


Figure 4.11: The installation directory under MacOS.

Configure Java Version

Especially if you have, for example, several JDKs installed or are not working on your own computer, it is very important that you check whether Eclipse is using the desired Java version. To check the configuration of the Java version, start Eclipse now. The development environment presents you with the ECLIPSE IDE LAUNCHER at startup. With the launcher you determine on which workspace Eclipse works. This is where Eclipse stores the Java source code and all project settings (section 4.3.10, »Eclipse Workspace«). Change the

path according to figure 4.12 and enter *Java-Projects/Java_with_Eclipse/Exercises* as the new workspace directory in the home directory of your user profile.

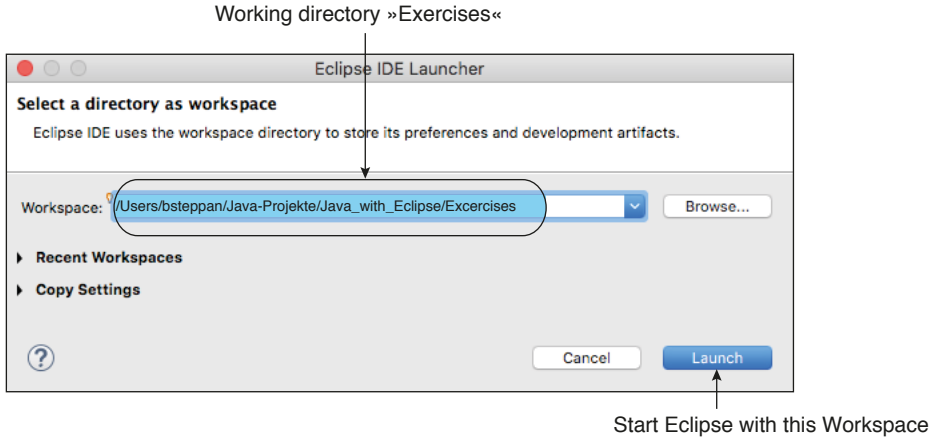


Figure 4.12: When you start Eclipse, enter the new working directory.

Then launch Eclipse with this workspace by clicking the LAUNCH button. Eclipse creates the directories previously entered under the WORKSPACE text field as a new workspace called *Exercises*. Now call the preferences of the development environment. On Linux and Windows, the corresponding menu command is PREFERENCES. It can be found in the WINDOW menu of Eclipse (Figure 4.13). On MacOS, the corresponding menu command is PREFERENCES and is located in the ECLIPSE menu of the development environment. You will use this dialog more often because it has a key function in setting up Eclipse.

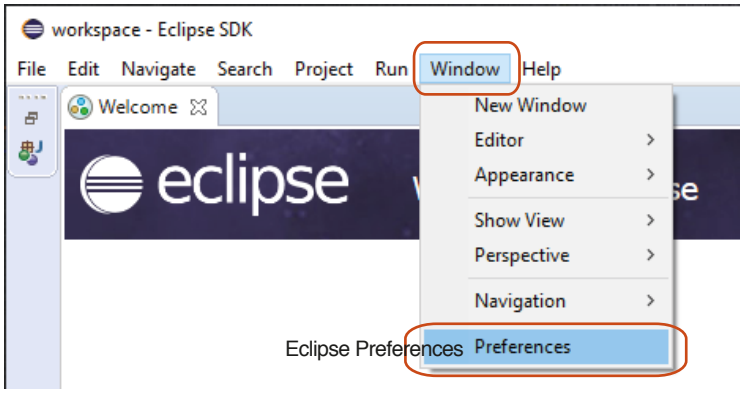


Figure 4.13: You can access the preferences under Linux/Windows via the »Window« menu.

After you execute the menu command, the PREFERENCES dialog for Eclipse preferences appears (Figure 4.14). Next, select the INSTALLED JREs setting, which is located in the tree structure on the left side of the dialog under the JAVA node. To find the setting, you can also simply type INSTALLED JREs in the search box at the top left (Figure 4.14 at the top left).

If a newer Java Development Kit with at least version 13 is selected in this dialog, as shown in the figure, you do not need to do anything. However, if this is not the case, select the Java version that you previously installed from the list. If this version is not listed, you probably have not set the search path to the Java Development Kit correctly. This is not tragic, but you will then have to manually search for the previously installed current Java version using this dialog. To do this, there is a button SEARCH... in the right part of the dialog. If you click on it, another dialog will appear, which you can use to navigate to the directory where the installed Java version is located.

Once the previously installed version is found, give it a catchy name, for example JAVA SE 13. Java SE means »Java Standard Edition« and is practically identical to the »Java Development Kit« (JDK). Set this version as the default in the dialog by placing the checkmark next to the name (Figure 4.14, top center). Configuring the Java version for Eclipse completes the basic installation.

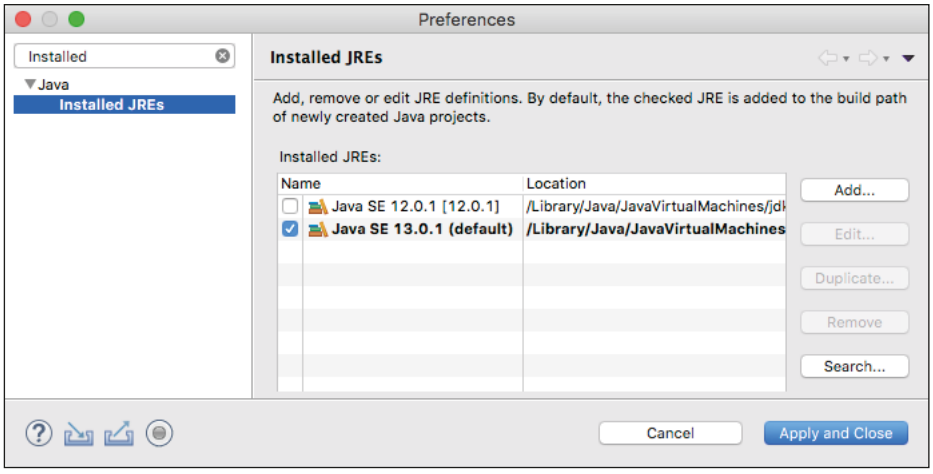


Figure 4.14: Eclipse should display at least Java 13 here.

4.2.4 Install Sample Programs

Downloading and configuring the sample programs completes the installation. You can find the sample programs including the bonus chapter as a zip archive on the Elektor Publisher homepage at *https://www.elektor.com/books/programming*.

Download the Sample Programs

Head to *www.elektor.com/books/programming* via an Internet browser. To find the archive file with the examples for this book, enter the author name STEPPAN or the book title GETTING STARTED WITH JAVA USING ECLIPSE in the “What are you looking for?” field indicated by the magnifying glass. Once the book marketing and resources page appears, scroll down to the DOWNLOADS tab to view the archive file(s) available for this book. Click on the .zip archive file with the sample programs, await the download process, and store the file locally on your computer for further processing.

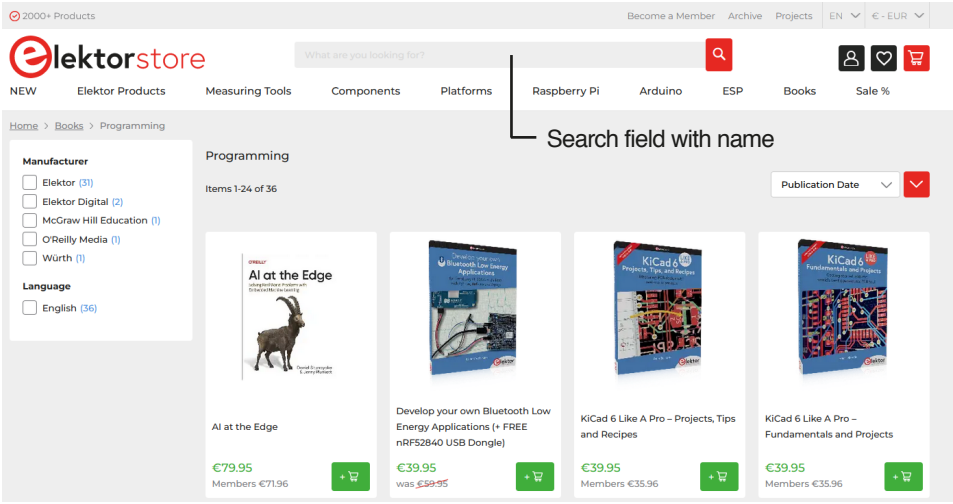


Figure 4.15: On the publisher's site you can get the sample programs and the bonus chapter.

Now download the zip file to your hard drive and unzip the archive to any directory. The zip file contains two directories: *sample programs* and *bonus chapters*. The *example programs* directory contains subdirectories corresponding to the chapters of this book. Each subdirectory of a book chapter contains another zip file. This zip file contains the packed Eclipse projects of the respective chapters.

Now move the complete directory *examples* into the already existing directory *entry_Java_Eclipse*. When you have moved the examples to this directory, the overall directory with all libraries and the workspace *exercises* should look like the figure 4.16 shows.

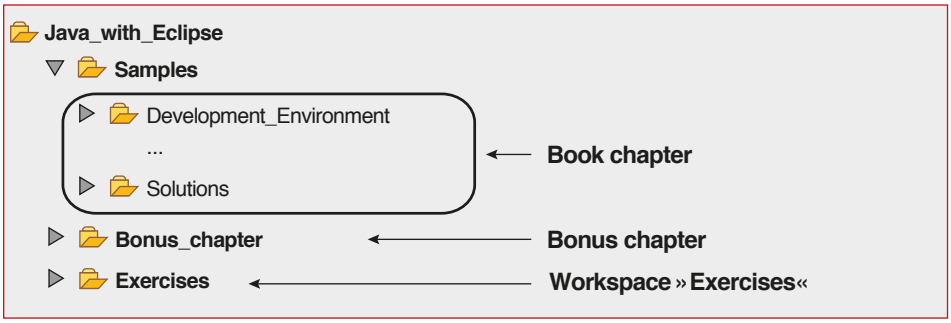


Figure 4.16: The structure of the sample programs follows the book chapters.

Importing the Sample Programs

Next, have Eclipse create a workspace for each chapter in this book. To do this, start Eclipse again with the workspace named *exercises*. So if you haven't changed the workspace in the meantime, clicking the LAUNCH button of the ECLIPSE IDE LAUNCHER again is sufficient. Once Eclipse is launched with this workspace, use the FILE → SWITCH WORKSPACE → OTHER menu command to switch to the *Development Environment* workspace located in the *Sample Programs* directory (Figure 4.17).

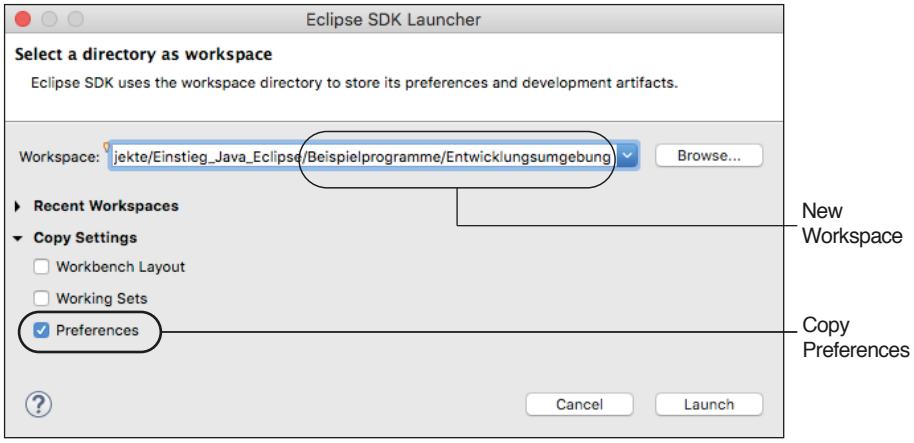


Figure 4.17: When you switch workspaces, you always adopt their settings.

When switching workspaces in this way, it is important that you have the option COPY SETTINGS → PREFERENCES set. This option will copy all the preferences from one workspace to the other. In each chapter directory there is a zip archive with the short name of the chapter with the extension *zip*. So for the chapter »Development Environment« the archive is called *Development_Environment.zip*, for the chapter »Program Structure« the file name is *Program_Structure.zip* and so on (Figure 4.18).

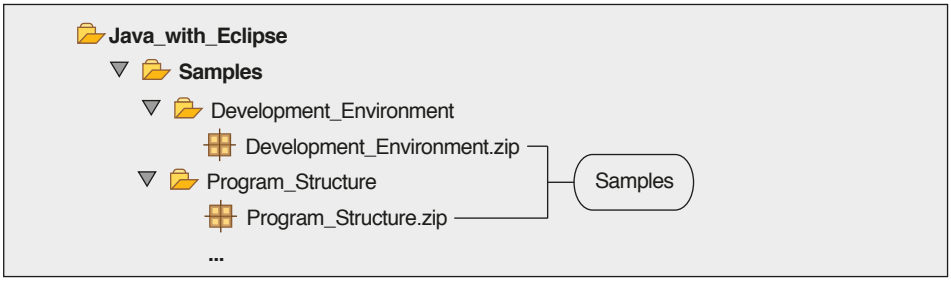


Figure 4.18: Each chapter directory contains a zip archive with the sample projects.

Next, import all the sample programs in a chapter. To do this, call the menu command FILE → IMPORT and select EXISTING PROJECTS INTO WORKSPACE in the import dialog. (Figure 4.19). Then click NEXT and use SELECT ARCHIVE FILE to select the directory of the chapter.

Once the archive has been read in, the Eclipse dialog shows the projects available in this archive under PROJECTS. Select all projects of the zip archive via the SELECT ALL button and start the import of the sample projects with FINISH. You have now imported all the sample programs.

In order to be able to start the programs, there is a so-called »Launch Configuration« in the *run* subdirectory of each project. To import this, call the menu command FILE → IMPORT again. In the import dialog under RUN/DEBUG select the command LAUNCH CONFIGURATIONS and switch to the next page of the dialog with NEXT (Figure 4.20).

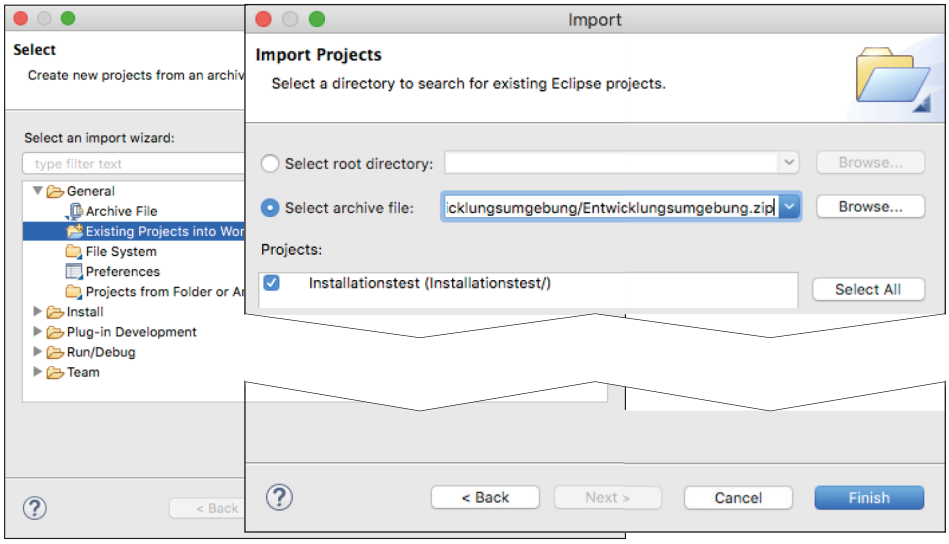


Figure 4.19: The first step in importing the sample programs.

The next page of the dialog contains a text box called FROM DIRECTORY. Use the BROWSE button to select the *run* subdirectory where the project's »Launch Configuration« is located. Once you have found the directory, click FINISH to load the configuration. Repeat the process for all projects in the zip archive and for all chapters of this book.

4.2.5 Installation Check

All workspaces are now created identically, all archive files are unpacked, and each project has its start configuration. When you have imported all projects and start configurations for all chapters, your directory should look like Figure 4.21.

Switch again to the *Development Environment* workspace using the FILE → SWITCH WORKSPACE → OTHER menu command. (Figure 4.22). You don't need to apply any settings this time because each workspace is already created identically.

Should Eclipse start with the WELCOME TO ECLIPSE view, switch to the Workbench by click-ing the WORKBENCH link in the upper right corner (Figure 4.23). Workbench is the »Eclipse« name for the visual working environment of the development environment. The welcome page is used to inform the Eclipse beginner about the development environment.

Eclipse switches to the Java perspective of the development environment by clicking WORKBENCH. Then call the RUN CONFIGURATIONS dialog. You can access the dialog using the RUN CONFIGURATIONS command from the RUN menu. (Figure 4.24).

In the left pane of the dialog you will find the subitem JAVA APPLICATION. If you have previously imported the LAUNCH CONFIGURATION named INSTALLATION TEST, it should be listed there. Select this configuration and then click the RUN button at the bottom of the

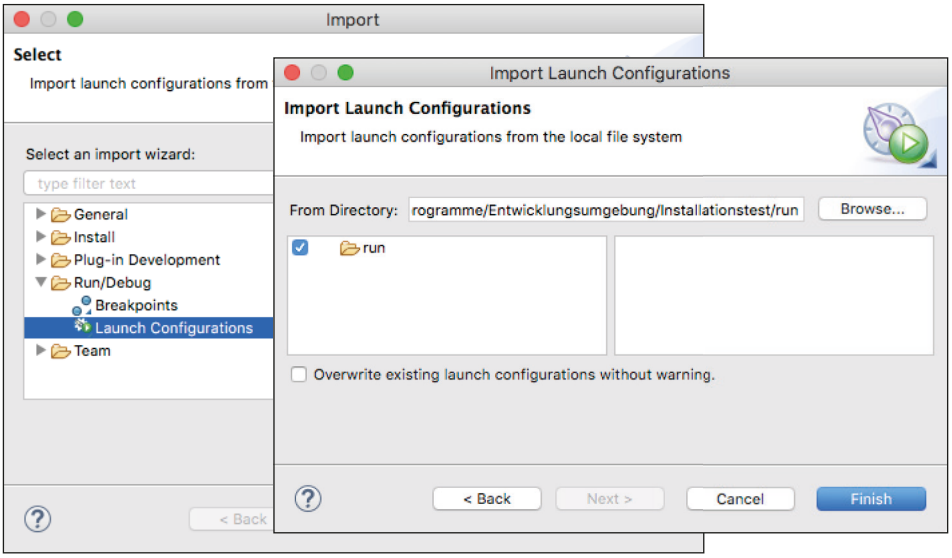


Figure 4.20: The import of the start configuration.

dialog. This should start the application. If the *installation check* program appears with the following success message, the installation has been completed successfully:

```
Okt. 01, 2019 12:16:42 AM programmingcourse.InstallationCheck main
INFO: Programm starts from: /Users/bsteppan/Java_Projects/Eclipse_Primer/
    SamplePrograms/Development_Environment/InstallationCheck
Okt. 01, 2019 12:16:42 AM programmiercourse.InstallationCheck main
INFO: The installation was successful. The detected Java version is 13.0.1
```

If the program does not appear or says that the installation was not successful, go through the installation steps in this section again to determine the error.

- Are you working with a 64-bit operating system?
- Is at least Java 13 installed?
- Is this a 64-bit version?
- Is the search path to the JDK set correctly?
- Is at least the Eclipse version »2019-12« installed?
- Are all examples installed?
- Have you imported the LAUNCH CONFIGURATIONS of the projects?

Chapter 26, »Frequent Errors«, will help you fix the most frequent errors.

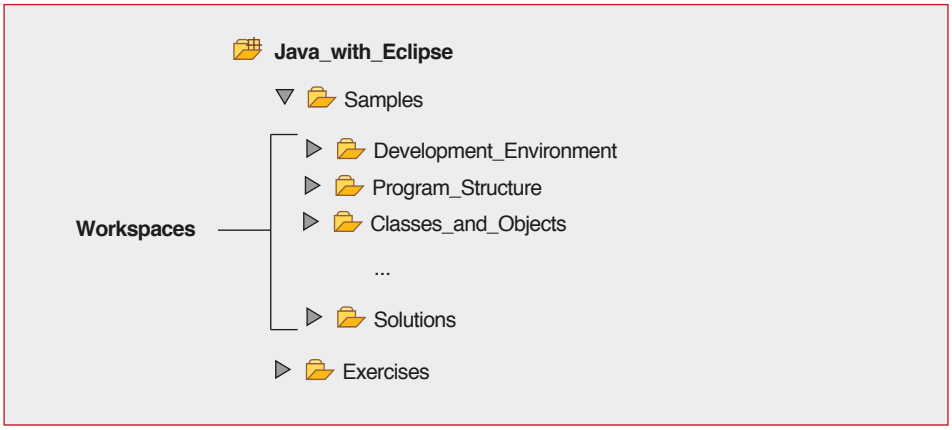


Figure 4.21: The directory of sample programs.

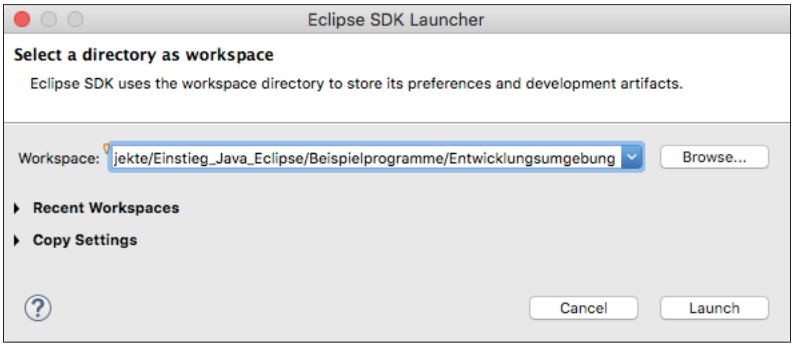


Figure 4.22: Switch to the »Development environment« workspace.

4.3 Eclipse Introduction

4.3.1 Overview

The Eclipse development environment essentially consists of seven elementary components: the menu at the top of the graphical interface, the toolbar directly below it, the project management (Package Explorer) on the left-hand side of the graphical interface, an editor with a window for the structure of the source code on the right-hand side and - more or less hidden in the background - the build system with its compiler, the debugger and the runtime environment (Figure 4.26).

4.3.2 Workbench

The Eclipse desktop is called the Workbench (Figure 4.27). It consists of views, editors and perspectives. The Workbench combines the individual components of the user interface such as menus, toolbars, status bars and windows. By using different perspectives, the

Figure 4.23: You will see this page when you start Eclipse for the first time.

Workbench can change completely at the push of a button. Perspectives allow the Workbench to be adapted to any work situation in the development of Java programs.

4.3.3 Perspectives, Views and Editors

A perspective is nothing more than a fixed window layout with a selection of certain views and editors. The Java"=development environment of Eclipse contains two important perspectives: the Java perspective and the debugging perspective. To better understand the impact of perspectives, launch Eclipse now if you have not already done so. When the Eclipse launcher appears, leave the »Development Environment« workspace and launch the Workbench with this working directory by clicking *launch*.

When the Workbench is visible, click the OPEN PERSPECTIVE icon of the toolbar (Figure 4.27). From the window that appears, select the DEBUG command. The command causes the Workbench to switch to the debugging perspective and change the arrangement and selection of windows accordingly as shown in Figure 4.28. You can see that the editor has remained in the center. But the toolbar has changed, and the views to the left and right of the editor are different.

Now exit Eclipse by closing the main window. On exit, the development environment saves the position and arrangement of the windows in the already mentioned workspace on disk (Section 4.3.10, »Eclipse Workspace«). Now restart Eclipse with the *development environment* workspace. What happens? Eclipse reads all the information from this workspace and opens the debugging perspective exactly as you last left the perspective.

A perspective like the Java"=Perspective is *only* a suggestion from the developers of the Eclipse"=Workbench on how you can work with the Workbench windows. To better adapt the perspective to your needs, switch back to the Java perspective. Now click on the tab of the OUTLINE window, hold down the mouse button and drag the window slowly to the bottom left with the mouse. As you drag it across the Workbench surface, a rectangle will

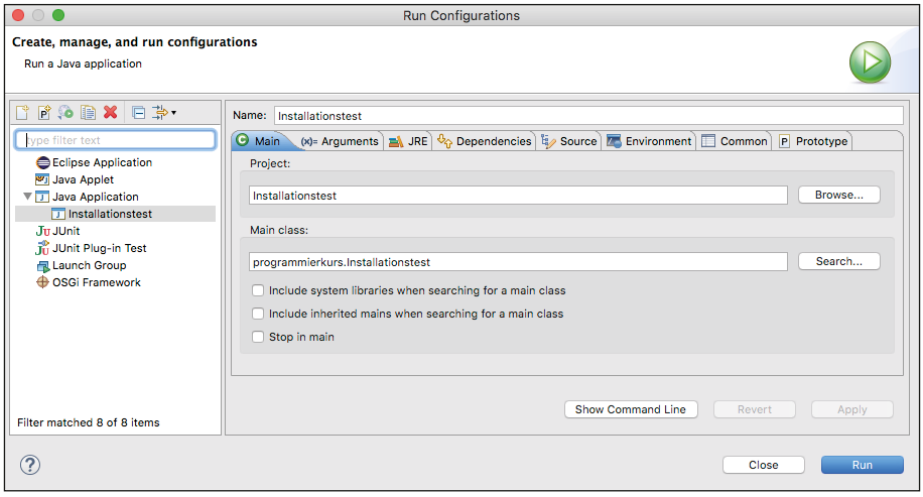


Figure 4.24: The startup configuration of the »Installation Test« program.

appear where you can dock the window. Dock the window below the Package Explorer by releasing the mouse button there.

Then right-click on the Java Perspective icon in the toolbar. From the context menu that appears, choose SAVE AS and in the window that pops up the Workbench, assign a name for this perspective such as PROGRAMMING COURSE (Figure 4.29). You have saved the modified window layout with the newly arranged OUTLINE window under a new name as a separate perspective.

Click the OPEN PERSPECTIVE button again and select JAVA (DEFAULT). The Workbench changes back to how it was initially seen. The toolbar has been given another icon for each new perspective. This happens whenever you call a perspective. If you want to remove the icons again, right-click on them and click CLOSE. The context menu contains two more interesting functions called RESET and CUSTOMIZE. The RESET command allows you to reset a perspective back to its initial layout. The CUSTOMIZE command lets you customize the perspective menu and toolbar commands. For the beginning I recommend you to leave all settings as they are, because they are chosen sensibly.

4.3.4 Package Explorer

The PACKAGE EXPLORER is the most important part of the Eclipse project management. The project management keeps your Java project together. With its help you configure where the Java source code is located, which Java version is used and which libraries. All information is stored in the workspace. In Java projects, you use the view called PACKAGE EXPLORER as the project management interface. If Eclipse is not already open, start the development environment with the last workspace named *Development Environment*. Then expand the components of the DEVELOPMENT ENVIRONMENT project in the Package Explorer. The explorer should look like the one shown in 4.30.

Figure 4.25: When everything is installed, you can jump to »Introduction to Eclipse«.

At the top is the folder icon for the project DEVELOPMENT ENVIRONMENT. Below that, you will discover an icon named JRE SYSTEM LIBRARY, a folder icon named SRC, and a folder icon named RUN. The JRE SYSTEM LIBRARY node shows you the Java Runtime Environment (JRE) configured for this project with its class libraries. In the SRC folder, Eclipse stores the project source code as Java classes with the extension *java*. I put the exported startup configuration of the project in the RUN folder.

In the right part of the PACKAGE EXPLORER, clicking on the downward arrow brings up a window menu. Here you can use the FILTERS menu to hide components that you do not want to appear in the PACKAGE EXPLORER. Another menu command is PACKAGE PRESENTATION. It causes packages to be presented either as a hierarchical tree or as a flat list. Again, I recommend that you leave all settings as they are.

4.3.5 Java Editor

Besides the project management, the Java editor is certainly the most important component of a Java"=development environment. With its help you develop your Java programs. The Java"=Editor consists of a sidebar on the left and one on the right as well as the view of the source code in the middle (Figure 4.31). Right-click on the sidebar on the left to see the context menu. It contains a wealth of functions, of which I will pick out just three: SHOW LINE NUMBERS, FOLDING and PREFERENCES.

Line Numbers

Now click on the command SHOW LINE NUMBERS. If line numbers were previously visible in the editor, they have now disappeared. Click again to make them reappear. Leave the line numbers on so that you can better follow the tutorials in the next chapters.

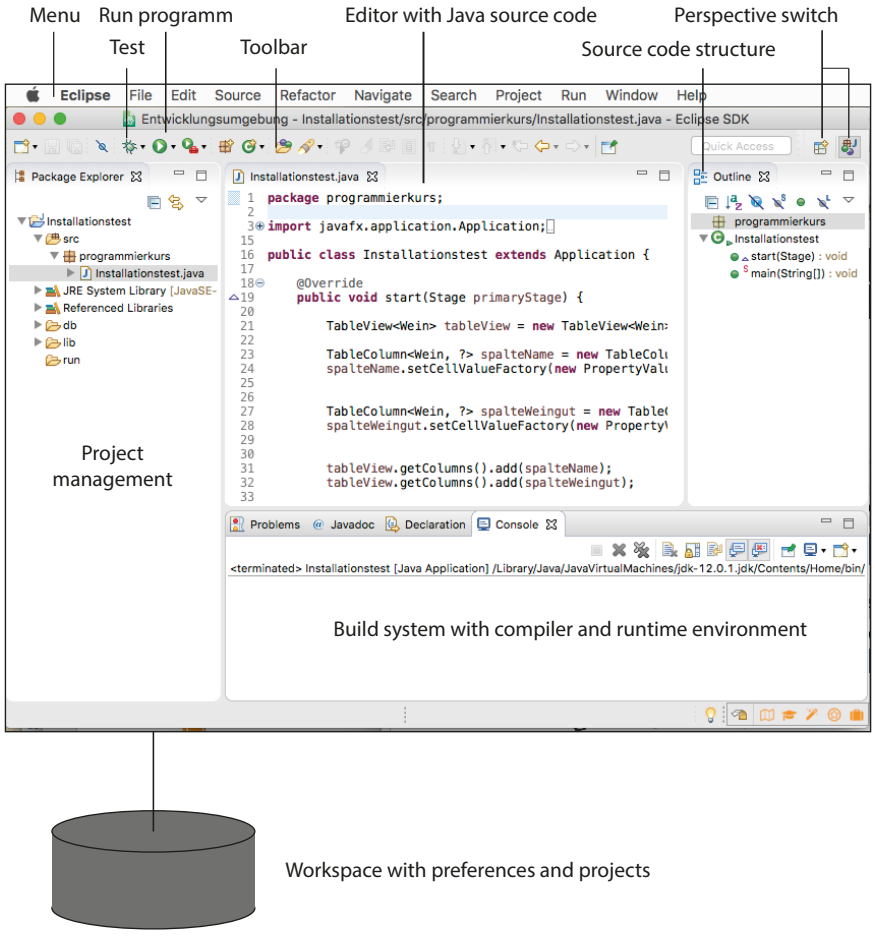


Figure 4.26: Eclipse at a glance

Source Code Folding

With the FOLDING command you can decide whether the editor displays the source code folded according to the language blocks. If the FOLDING is on, you can expand and collapse the blocks. You will also see small plus or minus signs to the right of the line numbers. If a minus sign appears, the area is expanded, if a plus sign appears, it is collapsed. If the area is collapsed, there is a jump in the line numbering of the editor at this point.

Preferences

The last of the three commands of the context menu is PREFERENCES. Click on it to get to the editor's preferences. Next, expand all parts of the tree on the left side of the dialog so that it looks like 4.32. Under the SPELLING item below TEXT EDITORS hides the spell checker (Figure 4.32). Make sure that it is turned off. It only slows down Eclipse and makes the editor mark any German-language comment in the source code as incorrect.

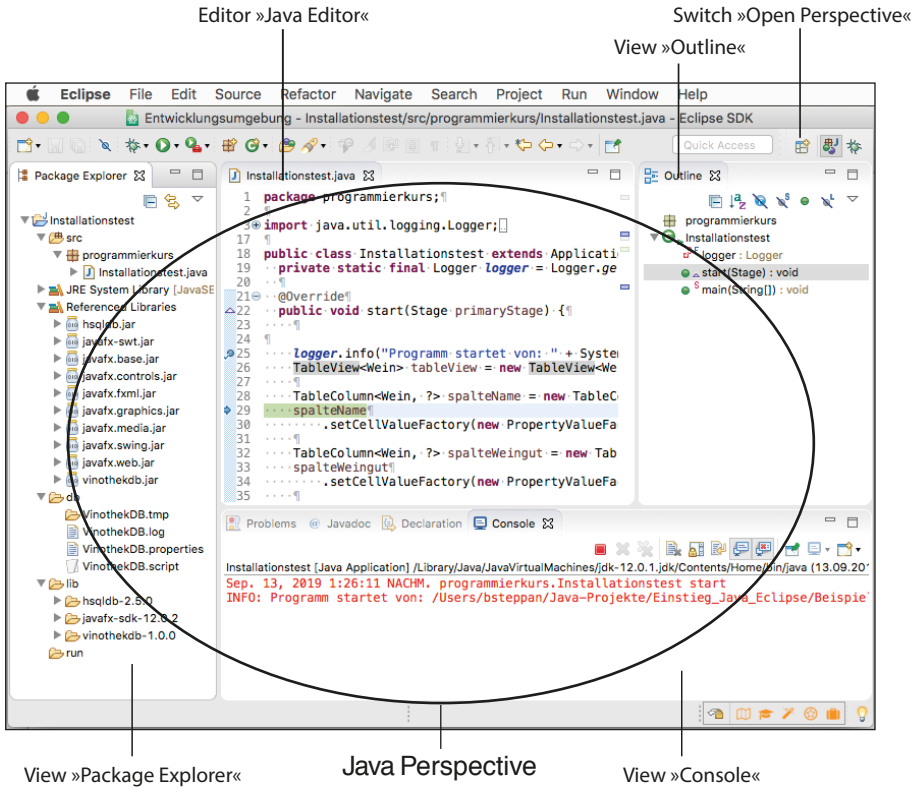


Figure 4.27: The Workbench's Java perspective.

Syntax Highlighting

Perhaps the most obvious difference in Eclipse's programming editor compared to a normal editor is the syntax highlighting. The Java Editor highlights all Java keywords such as *class* in bold and color. This increases the readability of the Java source code considerably. In the PREFERENCES dialog, click the SYNTAX COLORING subitem to access the syntax highlighting settings (Figure 4.33).

In the center of the window there is a field called »Elements«. If you click on one of the elements, you will see how the editor displays the element. The preselection can be changed by clicking on ENABLE. This will take you to the settings for all the elements and you can change the colors to your liking. Again, my recommendation is to change as little as possible at the beginning. You can now close the PREFERENCES dialog again.

Programming Help

The Java editor has a number of programming aids such as the programming wizard. Figure 4.34 shows how the wizard works using an object called *logger* as an example. When you type the dot after the object to invoke a method, the editor offers all the appropriate methods. You just select the method you want. The editor completes the rest. In addi-

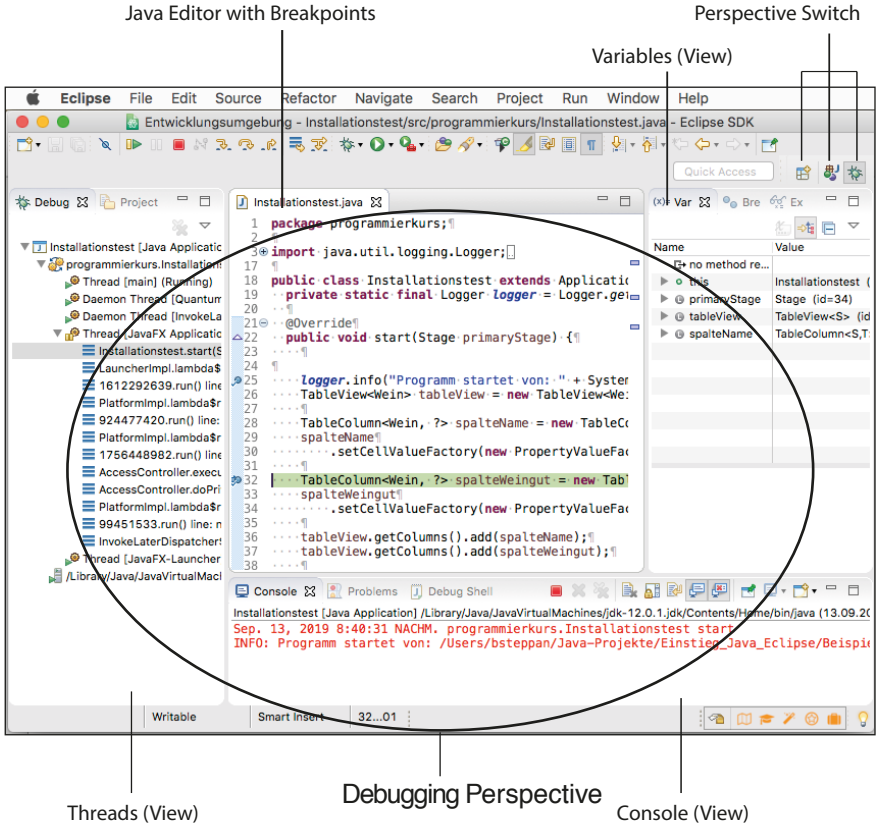


Figure 4.28: Eclipse's Debugging perspective.

tion to this programming wizard, the editor's programming help has other functions for generating code fragments, which I will present to you in the tutorials.

4.3.6 Code Formatter

The readability of the Java source code increases with clever formatting. For this reason, the source code can be formatted to match the structure of a Java program. To try out the formatter, start Eclipse now with the workspace *development environment*, if not already done. From the project, open the file *installationtest.java* via the PACKAGE EXPLORER by double-clicking on it. Remove the break between lines 24 and 25 so that the entire expression is on line 25:

```
24  logger.info("Program starts from: " + System.getProperty("user.dir"));
```

You can now call the code formatter from various places in the Workbench. The easiest way to do this is via the SOURCE menu. There you will find the command FORMAT about in the middle of the second section. If you click on it, you will see that the change is undone. You can set the code formatter to your liking using the Workbench preferences.

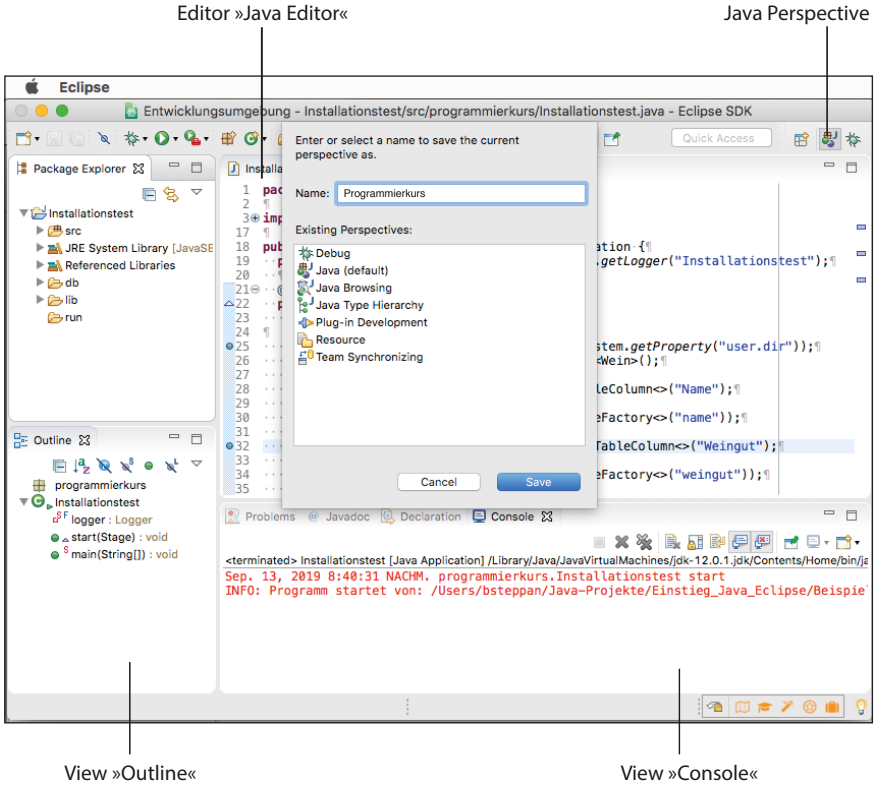


Figure 4.29: The first new perspective.

To do this, go to the preferences. On Linux and Windows, the PREFERENCES menu command can be found in the WINDOW menu of Eclipse. On the MacOS, the corresponding menu command is PREFERENCES and can be found in the ECLIPSE menu of the development environment. Enter »Formatter« in the search field of the dialog in the upper right corner to get to the right place right away. Select the code formatter below CODE STYLE in the JAVA section.

In the right part of the dialog you will see the EDIT button. If you click on it, another dialog opens with the ideas of the formatter. For example, you can set here whether the editor should insert tabs or spaces or both. The length of the indentation can also be configured here.

All in all, there are so many settings possible here that you can adapt the source code to (almost) any requirement. When you have made the settings as desired, assign a new name and save the setting with OK. Like all settings, the Workbench saves them in the Eclipse workspace. Like (almost) all settings, you can of course transfer them to other workspaces and export them as a separate file.

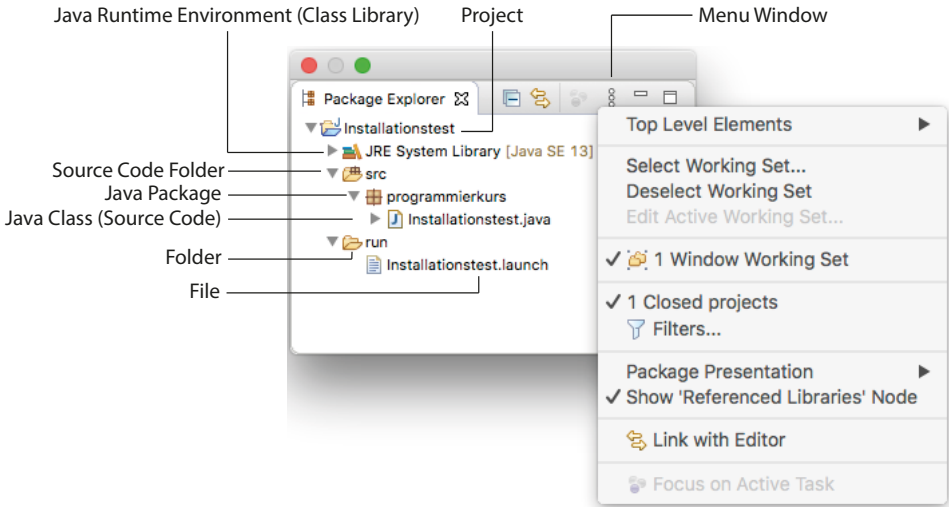


Figure 4.30: The Package Explorer gives you an overview of your project.

4.3.7 Build System

Eclipse's build system has two core parts: the build monitor and the Java compiler. The build monitor determines which Java classes can be compiled and passes this work to the compiler. Unlike a classic build system, the Eclipse compiler compiles the source code in the background by default. The way it works is that the Eclipse compiler always starts its work the moment you have saved a Java class and it is free of syntactical errors.

By compiling in the background, the build system ensures that you can almost always execute a program immediately. Especially for enterprise applications that consist of hundreds of classes and can have long compilation times, this is a plus over traditional development environments. So, as a rule, you don't do without this feature. If it should be necessary nevertheless once, switch it off over PROJECT → BUILD AUTOMATICALLY (Figure 4.36).

Like its JDK counterpart, the Eclipse compiler translates the Java source code into an intermediate format called bytecode. This bytecode can be executed by all virtual machines on Windows, Linux and MacOS. Chapter 20, »Development Processes«, goes into more detail about the compiler. Why a separate compiler? In the early days of Java, Eclipse's build system was a key advantage over other development environments. Today, computers and the JDK compiler have become so fast that this advantage is hardly noticeable.

4.3.8 Debugger

Another special feature of Eclipse is its integrated Java debugger. It is used to test Java programs and eliminate errors. If you have discovered an error and want to fix it, this does not require you to stop the program. Instead, you change the program while it is running. The modified class is automatically translated. The program can then continue to be examined

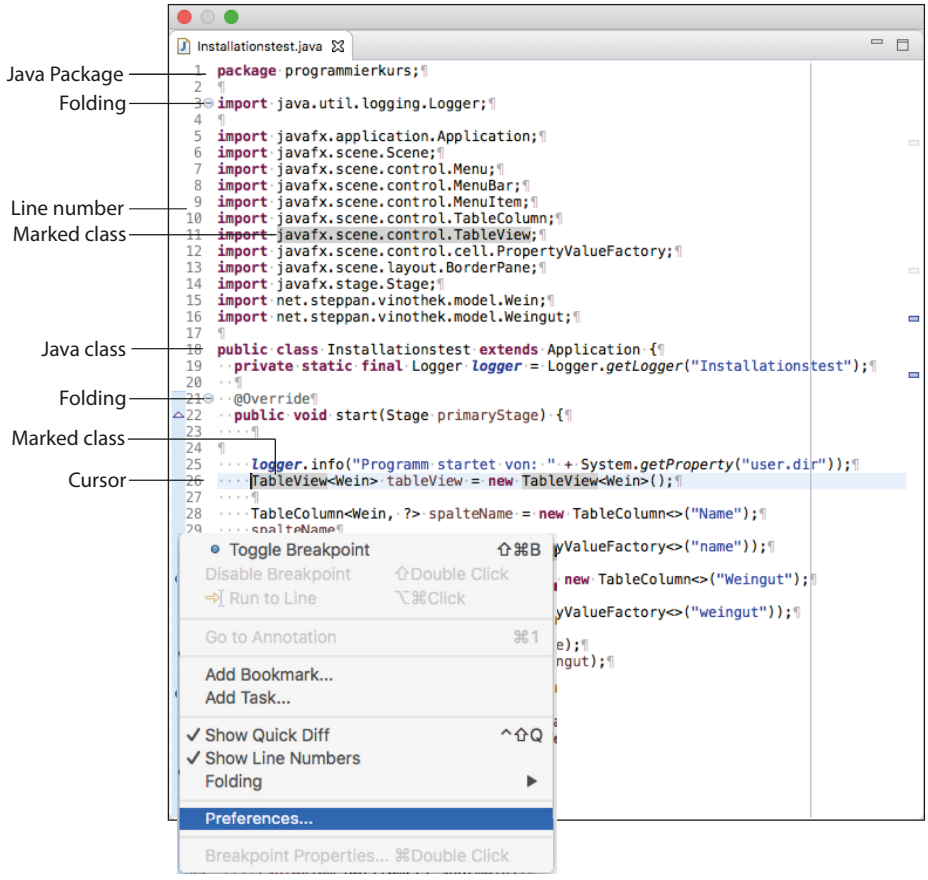


Figure 4.31: In the Java Editor you write your Java programs.

without interruption, since Eclipse replaces newly translated classes in the background (HotSwap replacement).

When the class is reloaded, the debugger executes the program again at the point before the error. Of course, this only works in cases where the changes are not too serious. If the changes are too serious, a HotSwap replacement fails. The program must be restarted in this case. Chapter 20, »Development Processes« goes into detail about testing a Java program with the Eclipse Debugger.

4.3.9 Modular Structure

Eclipse is a modular development environment. The Eclipse extensions are called plug-ins. Figure 4.37 shows the difference to a conventional development environment. A conventional development environment is shown on the left, and the Eclipse"=development environment is shown on the right. A conventional development environment consists of

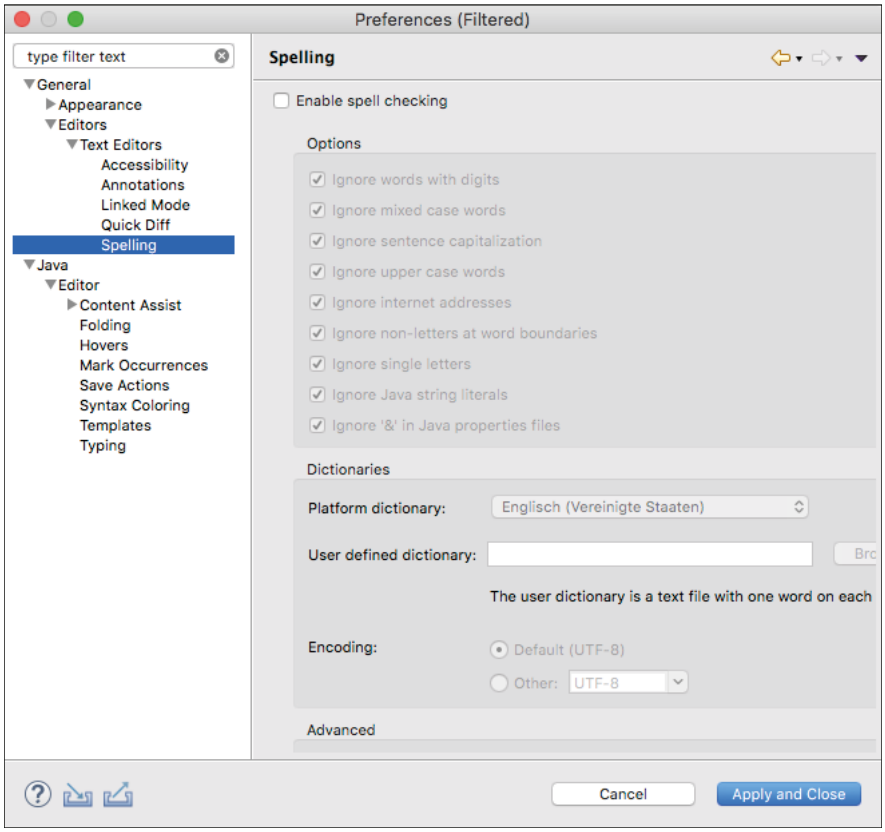


Figure 4.32: The spell checker should definitely be turned off.

a large, unchanging core. Individual tools such as the editors cannot be exchanged. In contrast, Eclipse consists almost entirely of plug-ins.

You can install your own plug-ins to modify Eclipse according to your needs. If you have special requirements, for example the development of web applications, you can download and install a suitable Eclipse package for this purpose. These packages are again nothing more than a collection of Eclipse Plug-ins for special areas of software development (section 4.2.3, »Eclipse Packages«).

4.3.10 Eclipse Workspace

The Eclipse development environment stores its program settings in a structured directory on the local hard disk, called the "workspace". The workspace is a special directory that contains all Java projects that have been saved in this workspace. In addition, Eclipse stores a number of hidden metadata there. They are hidden because they are internal files that the user should not change.

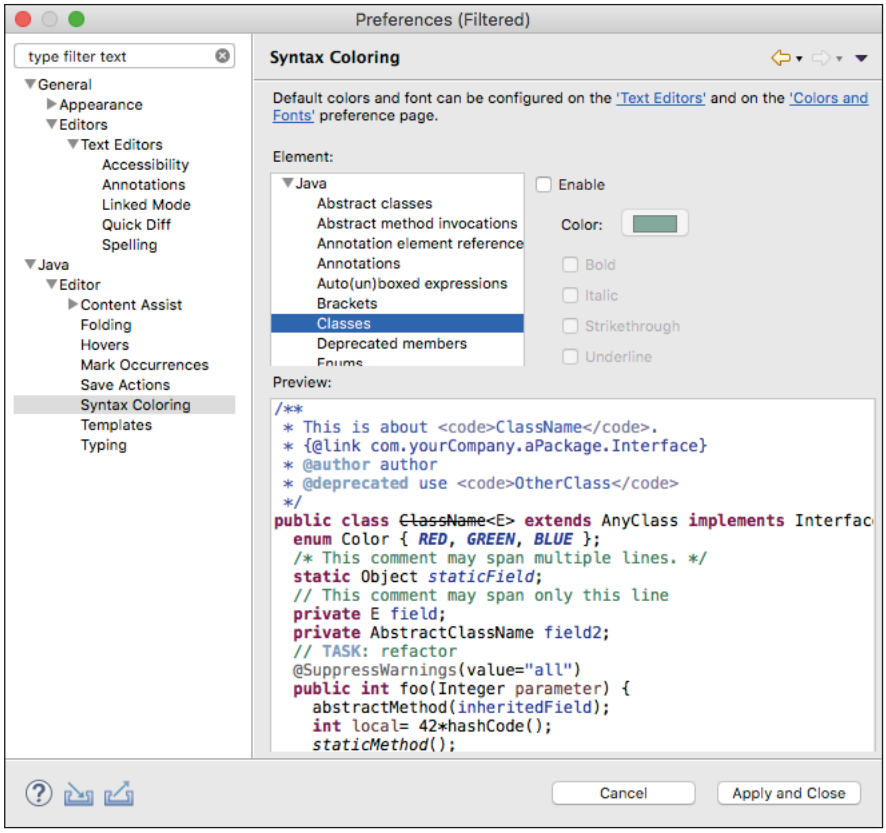


Figure 4.33: Syntax highlighting can be customized to your needs.

Eclipse stores a lot of information in the workspace: For example, Eclipse stores the Java version of the Java projects. Eclipse also stores which version of the development environment created the workspace. Last but not least, Eclipse stores which custom libraries are configured for this workspace (Figure 4.38). In a hidden lock file, Eclipse also keeps track of whether the workspace is already in the access of another Eclipse development environment. If this is the case, Eclipse refuses to access this workspace.

Setting up the workspace is a lot of configuration work in the first place. Eclipse can manage multiple workspaces with multiple projects, but can only open *one* workspace in each session. Therefore, it makes sense for your projects to work with only a few workspaces that are, for example, thematically separated.

A thematic separation could be for example to store different types of Java projects like web projects in different workspaces. Separate workspaces are also needed if, as in this book project, you have to work with a large number of small Java projects at the same time. Here a single workspace would be too confusing, because unfortunately the workspace cannot be separated by folders.

How to manage many different workspaces? As you have seen when creating the workspaces of the sample programs of this book, there are several functions to transfer all configura-

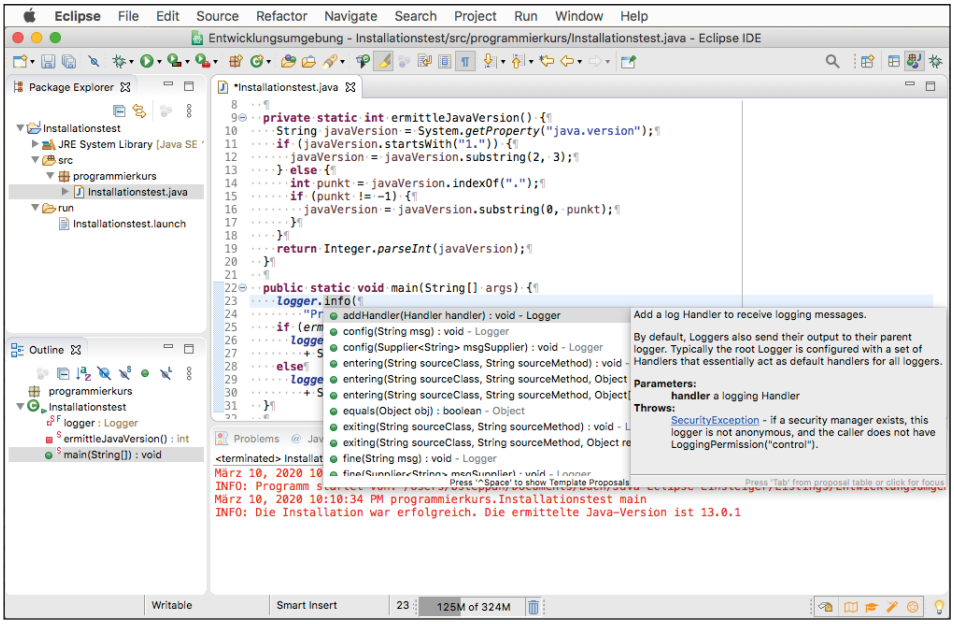


Figure 4.34: The Java Editor's programming help will assist you with manual development.

tions like libraries and Java runtime environments from one workspace to another. For example, the Eclipse IDE Launcher can be used to transfer workspace preferences (Figure 4.39).

You can also export the preferences using the FILE menu and its EXPORT command. When you access this export dialog, type »Preferences« in the upper left corner to get to the preferences export. If you want to export complete projects to a new workspace instead, enter »Archive File« as a filter. Projects can then be imported back into a new workspace via MENU → FILE → IMPORT, as you have already seen in section 4.2.4, »Importing the Sample Programs«.

4.3.11 Software Update

To keep Eclipse applications up to date after installation, the Eclipse IDE provides an update procedure. Through this update function, new versions of plug-ins or completely new plug-ins can be automatically or manually reloaded and removed again. This is usually done via an update website (cloud). But it is also possible to load updates via a drive or via an archive.

Eclipse is set to automatically check the Internet for updates and notify you when an update would be useful. You can find the settings under the PREFERENCES in the INSTALL → UPDATE section (Figure 4.40). Under the item AVAILABLE SOFTWARE SITES you can set which sites Eclipse should consider during the update. Of course, this requires an internet connection. Again, my recommendation is not to change anything in the preferences at the beginning, because they are chosen sensibly.

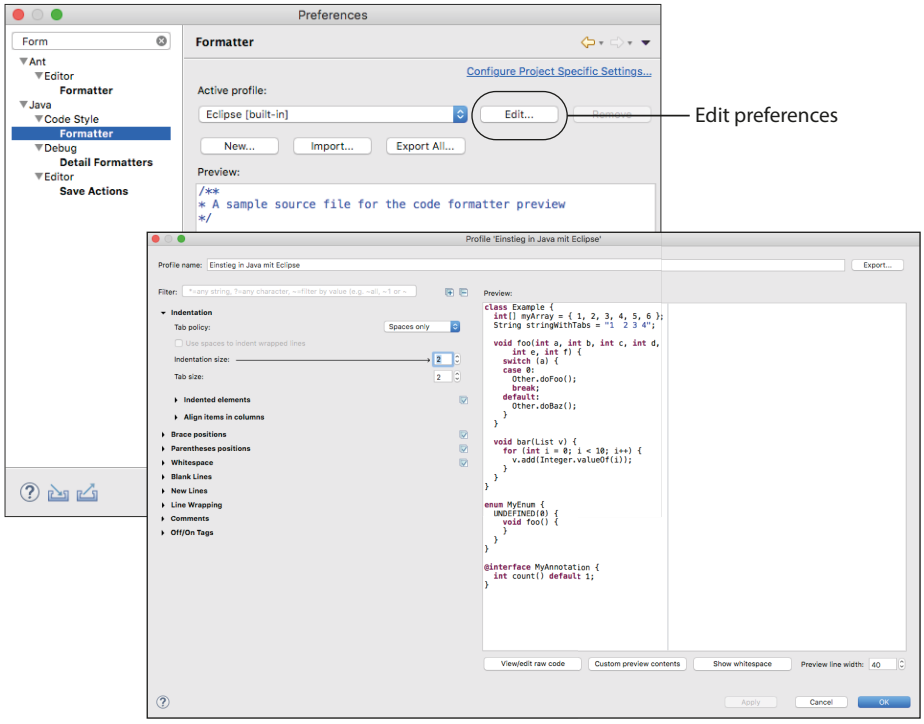


Figure 4.35: The formatter brings the source code into shape.

4.3.12 Help System

The Eclipse development environment offers both global help and context-sensitive help in each dialog. In addition, there are various tutorials that guide you step by step, for example, when creating a project. You can access these tutorials via the menu command HELP → WELCOME. The menu command shows the welcome page with which Eclipse greeted you after the first start of the development environment. The welcome page can be operated like an internet browser. It consists of individual links. You can use the toolbar in the upper area to scroll back and forth.

On the top right of the home page you will find a link to tutorials. The link SAMPLES will take you to more advanced examples. Note that both the tutorials and the sample programs are aimed less at Java beginners than at professional Eclipse"=developers. They are very much focused on developing programs to advance the Eclipse platform. Therefore you will also find Eclipse special topics like the development of SWT and RCP programs. SWT is the standard widget toolkit of the Eclipse platform. RCP is the rich client platform of Eclipse. If you want to return to the Eclipse working environment after studying the welcome page, click on the WORKBENCH link, which is always located in the upper right corner of each page.

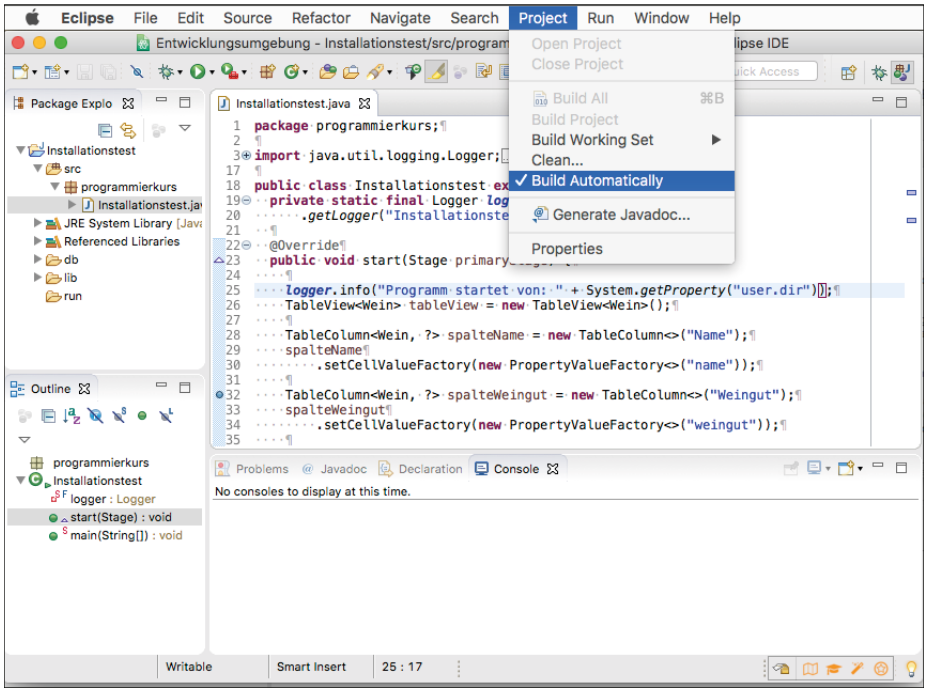


Figure 4.36: The Eclipse compiler translates a Java program in the background.

4.4 Summary

Eclipse is a development environment tailored to professional software development. The Eclipse IDE differs from other development environments by a number of special concepts. These include the modular structure with plug-ins and the layout of the interface with perspectives, views and editors. Other special features of Eclipse are the build system and the concept of the workspace.

Eclipse seems too complicated for many beginners because there are so many settings. Therefore, especially at the beginning, it is important to understand the basic concepts of Eclipse in detail: the structure of the interface, the build system, the workspace and the configuration of projects. The tutorials in the next chapters deepen the information in this chapter and show you how to apply the Eclipse concepts.

4.5 Literature

Wikipedia: Software repository; *https://en.wikipedia.org/wiki/Software_repository*

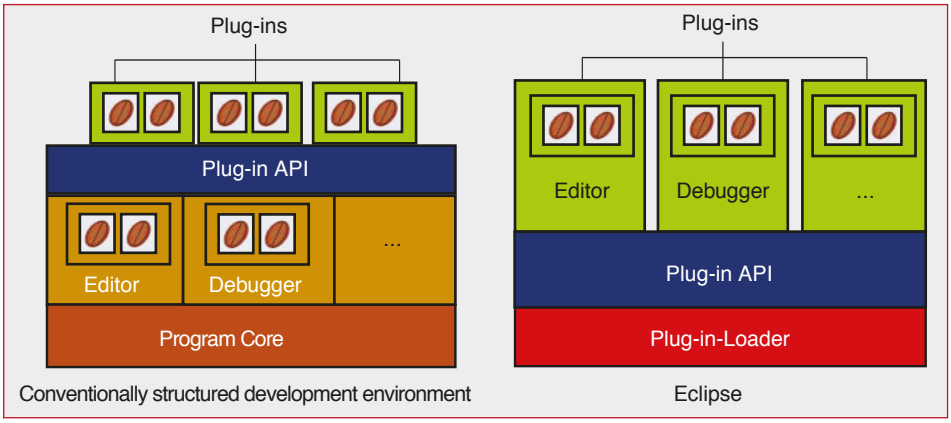


Figure 4.37: Comparison between a conventional and the Eclipse development environment.

4.6 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge: *https://www.programmingcourse.net/books/java_with_eclipse/development_environment* There you will also find the solutions to the exercises.

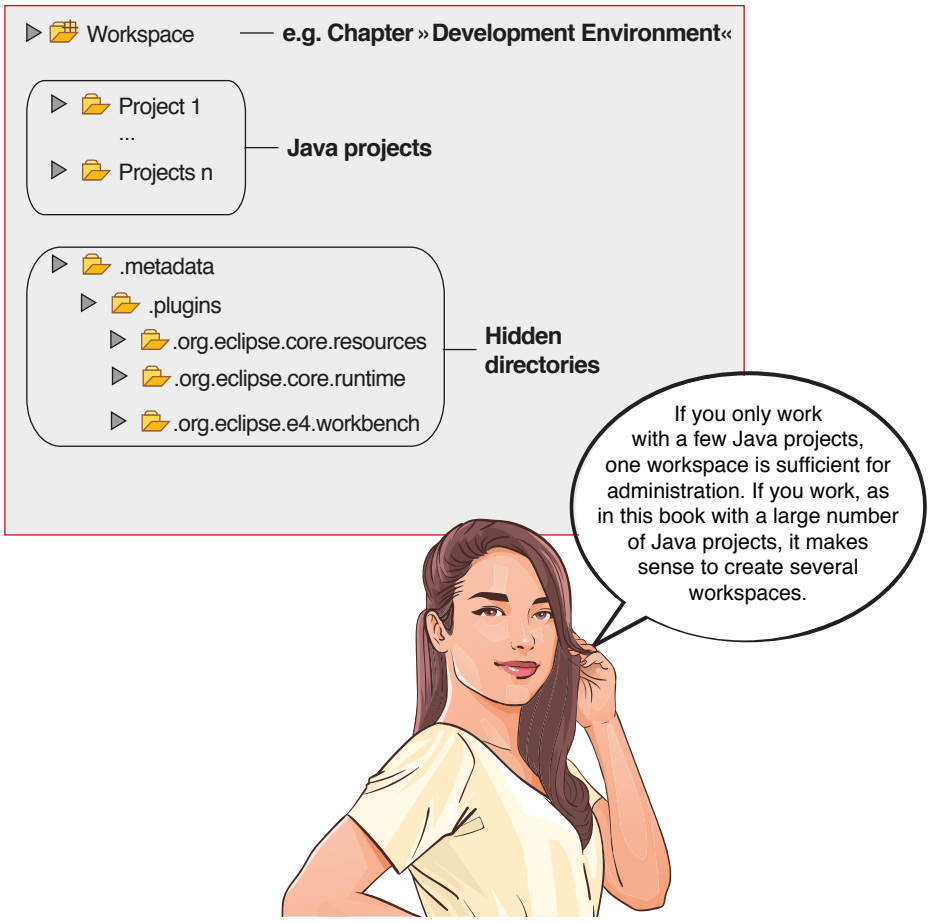


Figure 4.38: The structure of the Eclipse workspace.

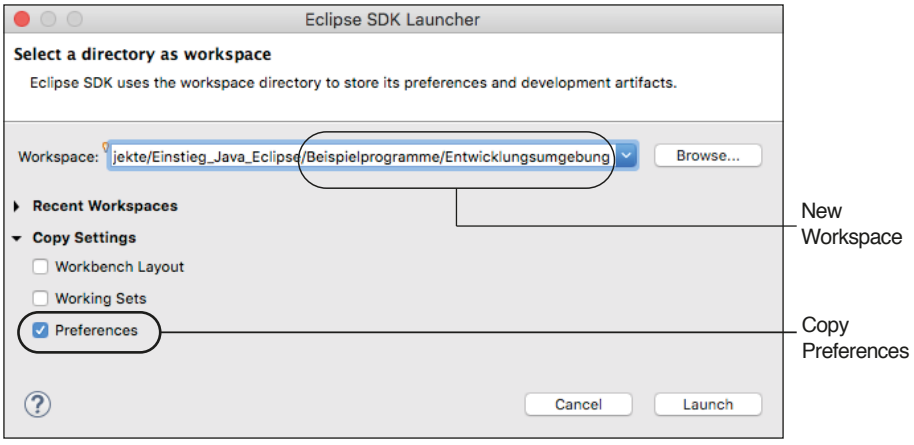


Figure 4.39: When you change the workspace, you can transfer the settings.

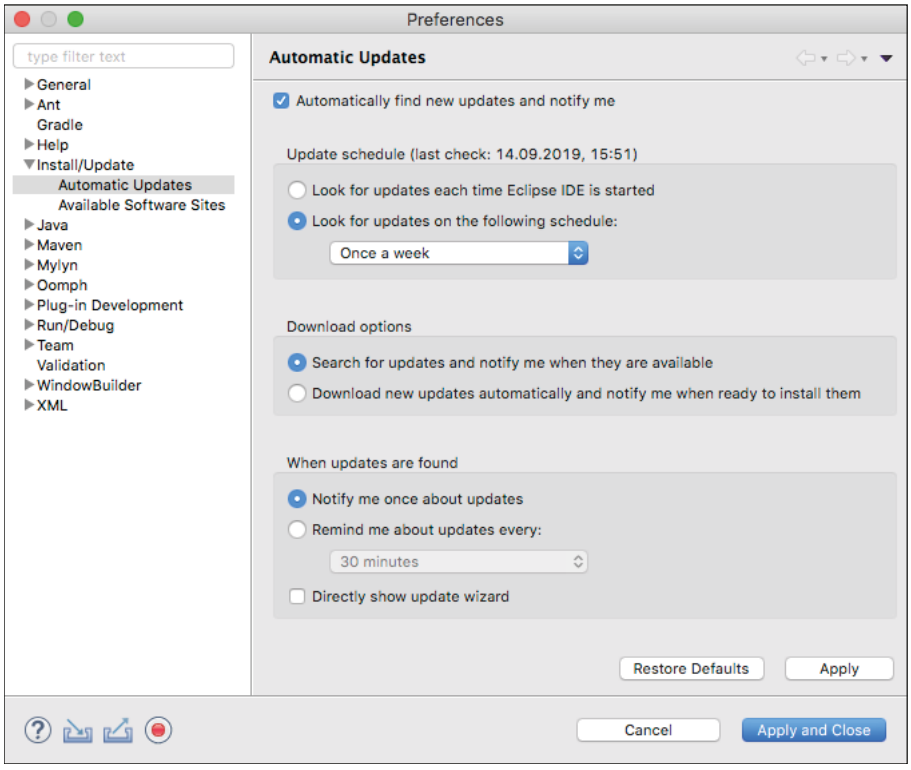


Figure 4.40: In the »Install/Update« section you define the settings for updates.

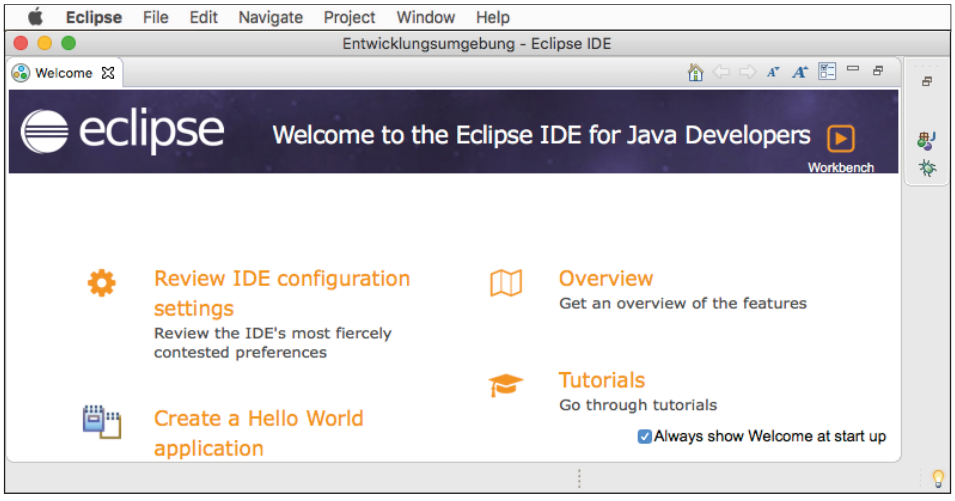


Figure 4.41: On the welcome page you will find an Eclipse overview and tutorials.

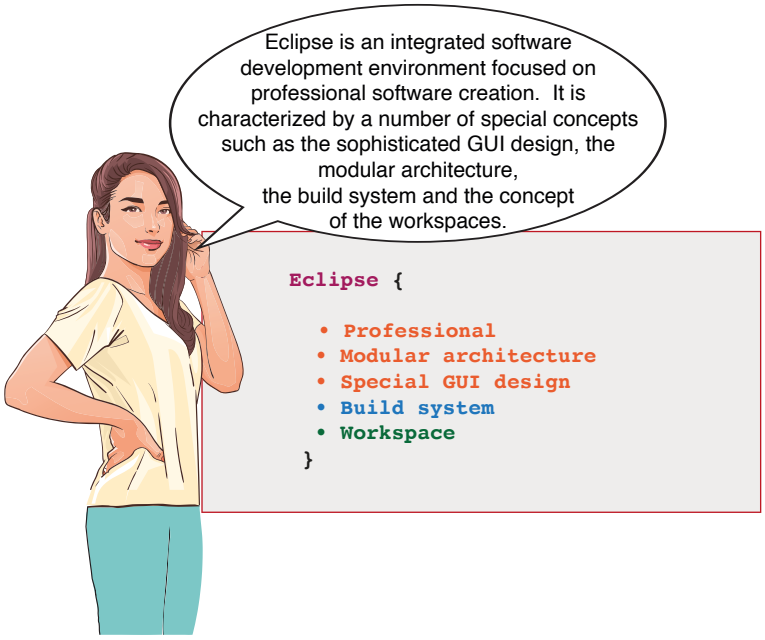


Figure 4.42: Julia summarizes this chapter.

PART II

Java Language

After the first part of the book has laid the programming foundations, this part is about the basics of the Java language. You will get to know all the important elements of the Java programming language using smaller examples. The arc spans from the basic structure of a Java program, variables and constants, statements to classes and objects. Methods, operators, branches and loops illuminate the dynamic side of Java programming. The book section ends with packages and modules as well as a chapter on error handling and documentation.

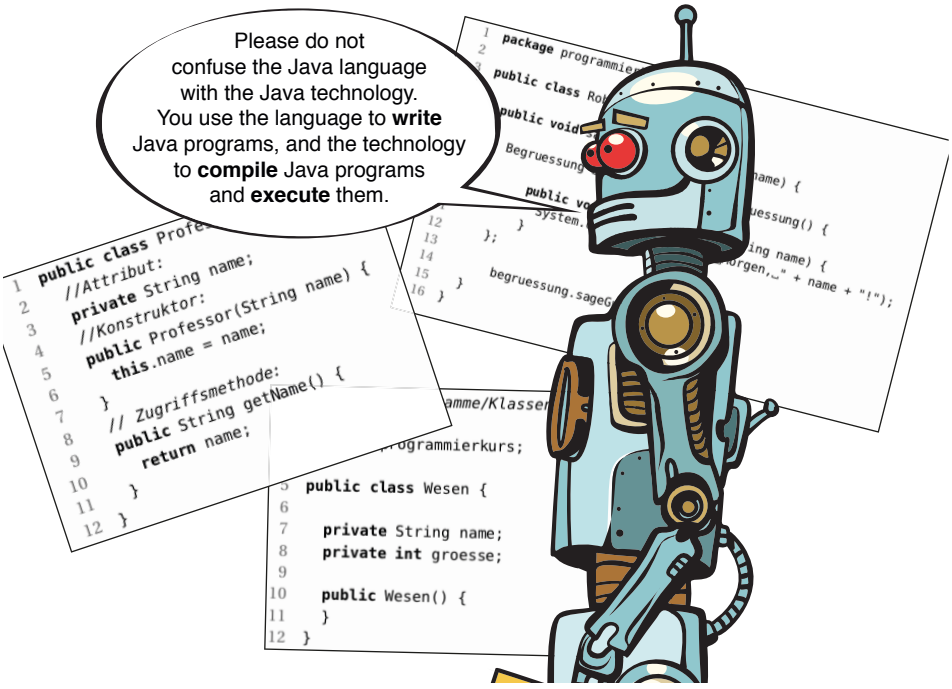


Figure 4.43: Roland snuck into this section of the book.

Each of these chapters first presents how to use the respective language elements in Java programs. Afterwards, exercises show you how to develop the previously shown programs step by step with Eclipse. The exercises at the end of the chapters help you to check whether you can apply what you have been shown before.

5 Program Structure

5.1 Introduction

Java and the Eclipse development environment are now installed with all the necessary tools. The basics of object-oriented programming are also laid. It is time for the first simple Java programs. This chapter shows you how Java programs are generally structured. You will learn the first basic concepts like classes and objects, methods and statements as well as packages and comments in the context of the Java language.

```
// Examples/Program_Structure

package programmingcourse;

class ProgramDemo {

  public static void main(String[] args) {

    Person roth;

    roth = new Person ("Professor Roth");

    System.out.println("The person is call
      + roth.getName());

  }

}
```

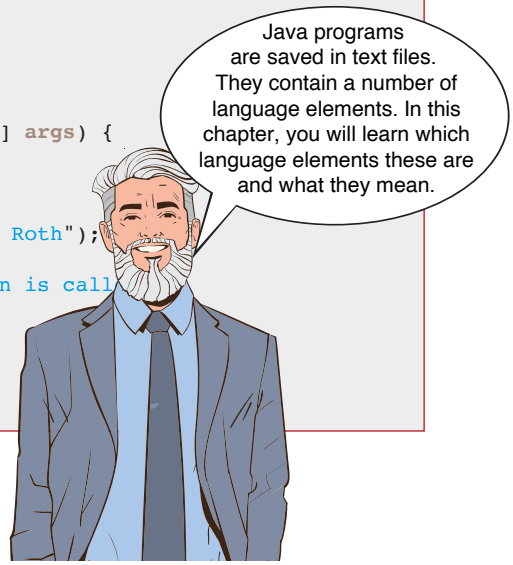


Figure 5.1: Professor Roth will guide you through this chapter.

The tutorial at the end of the chapter show how to create Java projects and Java programs using the Eclipse development environment, how to define classes and methods, and how to compile and run programs using Eclipse.

5.2 Overview

Java programs are described in one or more text files. The name of the class gives the text file its name. The text file must also have the extension *java*. Example: So, if you want to write a class called *ProgramDemo*, you must store it in a text file called *ProgramDemo.java* (Figure 5.2).

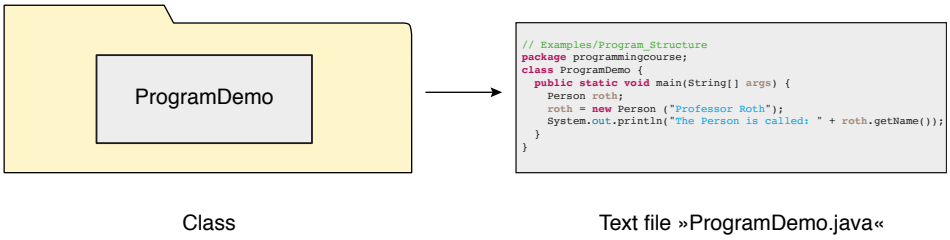


Figure 5.2: The class »ProgramDemo« is stored in a text file with the same name.

When you create a new class named *ProgramDemo* in the Eclipse IDE, the development environment automatically creates a text file named *ProgramDemo.java*. The listing 5.1 shows how such a class might look like after some programming.

Listing 5.1: The program »ProgramDemo« as a file »ProgramDemo.java«.

```
1  //Examples/Program_Structure
2
3  package programmingcourse;
4
5  public class ProgramDemo {
6
7    public static void main(String[] args) {
8      Person roth;
9      roth = new Person("Professor Roth");
10     System.out.println("The person is called " + roth.getName());
11   }
12 }
```

The sample program *ProgramDemo* presented here produces the following output:

```
The person is called Professor Roth
```

Please note with the listings that the line numbers do not belong to the program. They only make it easier for me to refer to certain code passages. Therefore, most of the listings in this book are printed with line numbers for explanation.

5.3 Language Elements

In figure 5.3 you can see an overview of the components of the previous program example. Some elements you already know from the previous chapters, such as class or method. Other elements like package, declaration, start of block or end of block are new.

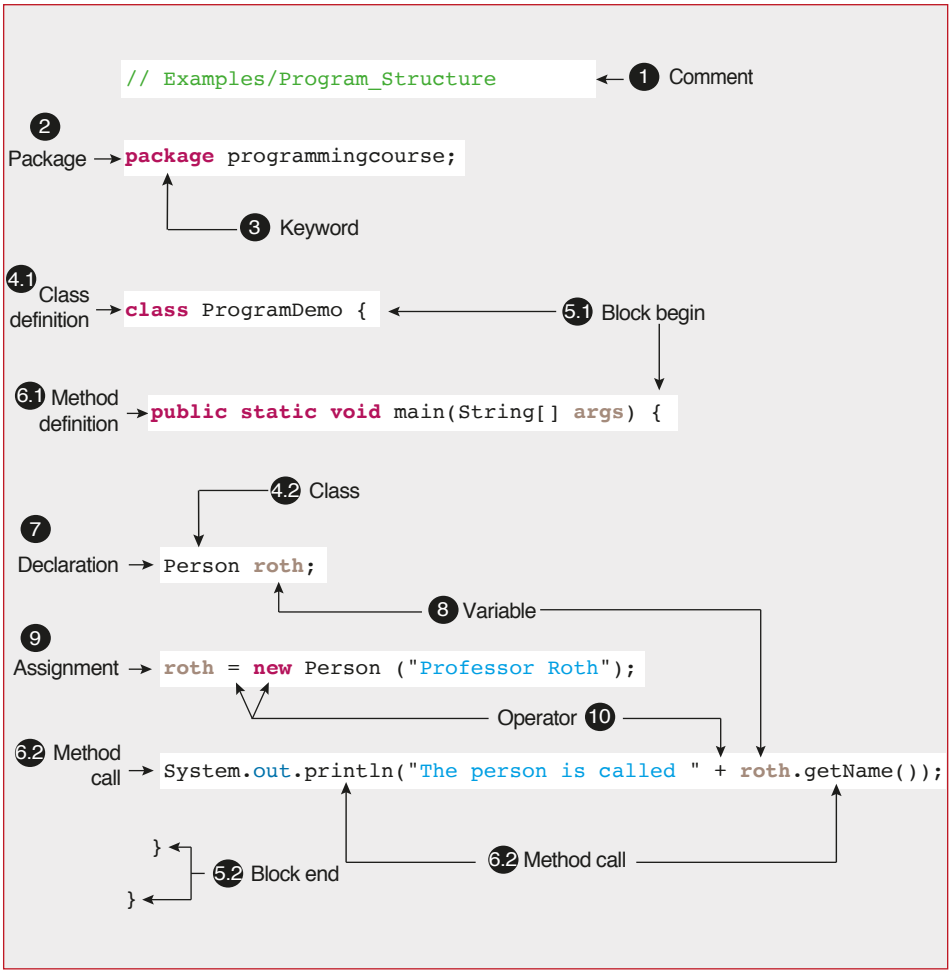


Figure 5.3: Overview of the most important Java language elements.

To give you an overview of the structure of a Java program, I would like to briefly introduce the most important Java language elements on the next pages. All elements will be explained in detail later in separate chapters.

5.3.1 Comments

The program example starts with a comment in the first line (Figure 5.3, Point 1). Comments are used to document the program. A good programmer comments a lot for the understanding of a program – not only for others who work on the program, but also for himself. Comments facilitate thus the understanding of the program.

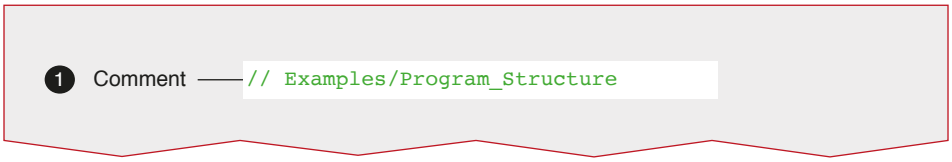


Figure 5.4: Comments document Java programs.

In this case, the comment says that it is an example of the chapter »Program Structure« of this book. The documentation of a Java program ranges from a comment of a single line of code to complete system documentation. Chapter 18, »Documentation«, introduces you to the possibilities of documenting a Java program.

5.3.2 Packages

The keyword *package* in this program denotes the package *programmingcourse* (Point 2). The reserved word *package* is one of the many Java keywords (Section 5.6, »Reserved Keywords«). Packages structure Java programs. They package the classes of a Java program into larger units, hence their name.

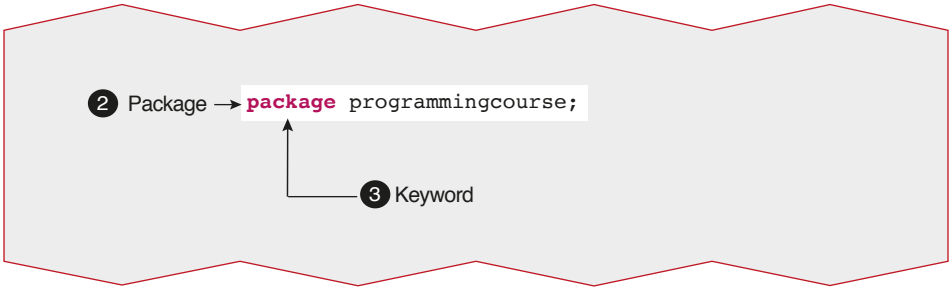


Figure 5.5: Packages pack one or more classes.

This book shows packages in graphical images as folder symbols. If you have the book in color print, for example as PDF, packages in graphics are shown in light yellow. For more on packages, see chapter 16, »Packages and Modules«.

5.3.3 Classes

Chapter 3, »Object-Oriented Programming«, was about classes that represented »natural« persons like Anna, Julia and Robert. In this example, on the other hand, a »technical« class

called *ProgrammDemo* plays the main role. The class *ProgramDemo* is a Java program. It creates an object of the class *Person*, the well-known Professor Roth. It then outputs the name of the professor and then ends (Figure 5.6). How is the program structured?

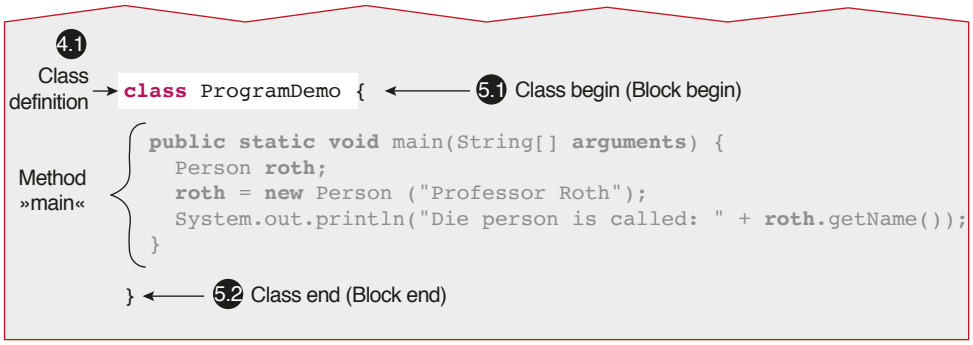


Figure 5.6: Classes begin and end with a curly bracket.

The program starts with a class definition as seen in Point 4.1 of Figure 5.6. The class definition introduces the keyword *class*, which is followed by the name of the class - here it is the name *ProgramDemo*. Class definitions always start with an opening curly bracket (Point 5.1) and end with a closing curly bracket (Point 5.2). The area between two curly braces is called a block and usually contains various statements and methods. The class *ProgramDemo*, however, consists only of the method *main*.

If you reduce the program named *ProgramDemo* to its core, the result is the very simple program structure shown in Figure 5.7: The program consists of a class called *ProgramDemo*, which has a method *main*. This method contains some instructions that the program executes.

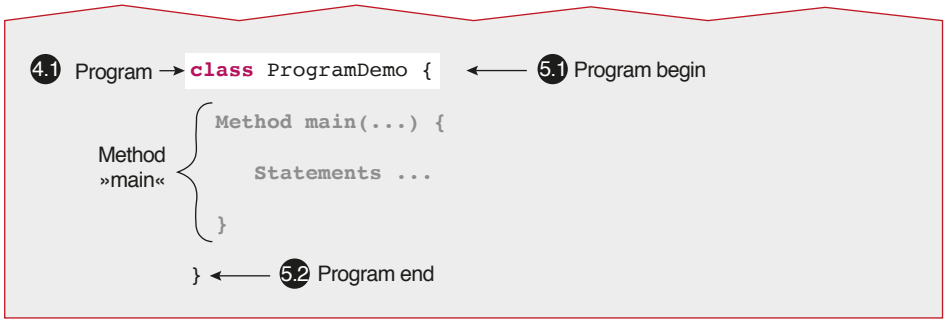


Figure 5.7: A program is a class with a method called »main«.

5.3.4 Methods

The method called *main* is introduced with the keywords *public static void*. Forget about the keywords *public* and *static* for now. They only distract from the structure of the method and will be covered later in this book. The actual beginning of the method is introduced by

the method name *main*. This is followed by a pair of parentheses containing the parameter *String[] arguments*, which you also ignore for now. If you reduce everything to the actual core, you get a very simple picture of the structure of a method (Figure 5.8).

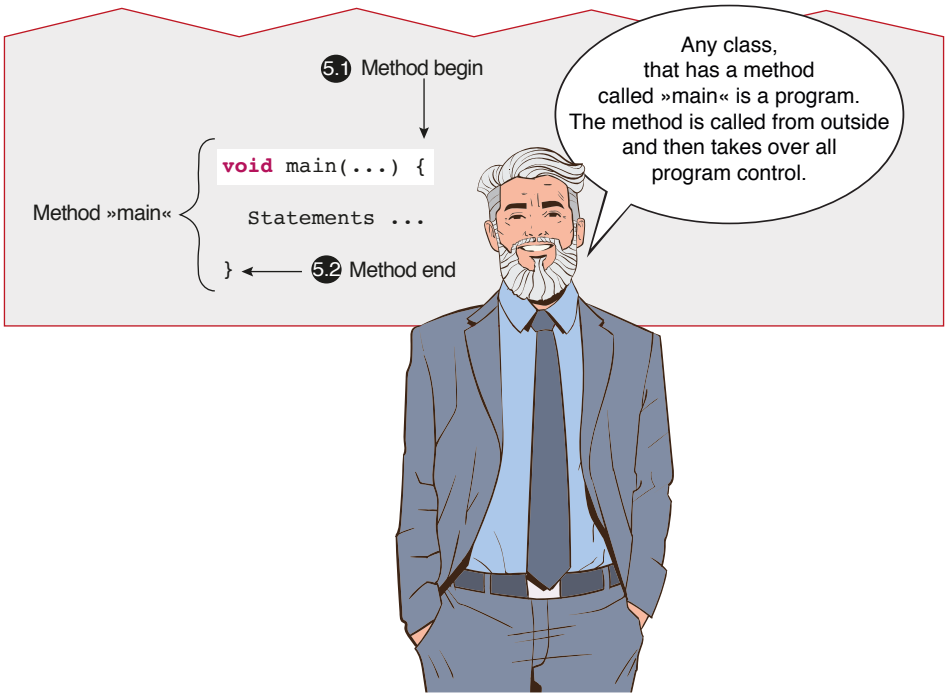


Figure 5.8: Every class with the method »main« is a program.

As with the class definition, the method begins with an opening curly bracket and is terminated by a closing curly bracket. In between there is a block of statements. The name of the method *main* is preceded by the keyword *void*. The background: every Java method must either have a return value or specify that it does not return a return value. In the case of the method *main*, no return value is required. Therefore, the method name is preceded by the keyword *void*, which means »no return value«.

However, the method *main* is not just any Java method. It is special among all Java methods: any Java class that has such a method is a Java program in its own right. The method is called from outside the Java program and then takes control of the program. Thus, the instructions contained within the method determine the entire program flow.

5.3.5 Statements

You can compare Java statements to a natural language sentence that contains commands. Such a statement might be, »Output the name of Professor Roth using the Java function *println()* of the class *System*«. There are many different statements in Java programs, such as declarations, assignments, and output statements.

Declaration

The first statement within the method *main* is a declaration (Point 7). Through it the program »declares« what data type the variable *roth* should be (Point 4.2). Since the data type is the class *Person*, the variable in this case is an object (Point 8).

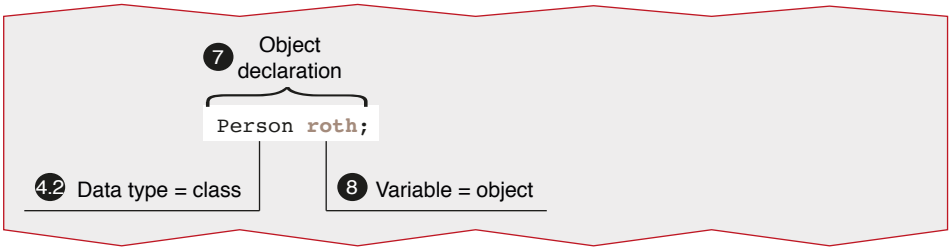


Figure 5.9: The declaration determines to which data type a variable belongs.

You may wonder at this point if the class *Person* has been defined before. The declaration actually requires an existing class definition and this is located in another text file called *Person.java*, which I have withheld from you so far (Figure 5.10). The exact structure of this class should not be of interest at the moment.

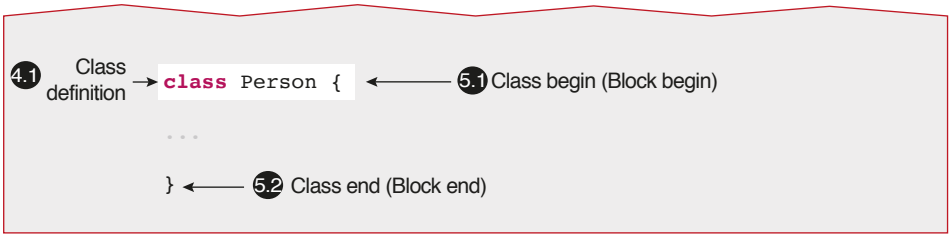


Figure 5.10: The class »Person«.

Assignment

After the declaration of the object named *roth* follows an assignment using the assignment operator to the previously declared variable named *name*. To the right of the assignment operator, the program creates an object of type *Person* and passes it the name *Professor Roth* in the form of a string of characters. You already know that a constructor is necessary to create objects. At this point, the Java program calls it (Figure 5.11).

Now we have to return to the class *Person*. This class contains the constructor that was just called. The constructor is used to construct an object of type *Person*, in other words, to create it. With the help of the constructor, the program is also able to specify the name of the person. For this purpose, the program passes the string *Professor Roth* to the constructor.

A character string in Java is of the type *String*. This is a class already predefined in Java, which the second part of the book introduces to you in more detail (Chapter 22, »Class Libraries«, section 22.3.1, »Class »String««). The string passing consists of the variable *name*, which is of the type *String*. The passing is done as in a declaration in the format data type followed by the name of the variable. Explanation of the exact flow and meaning of the

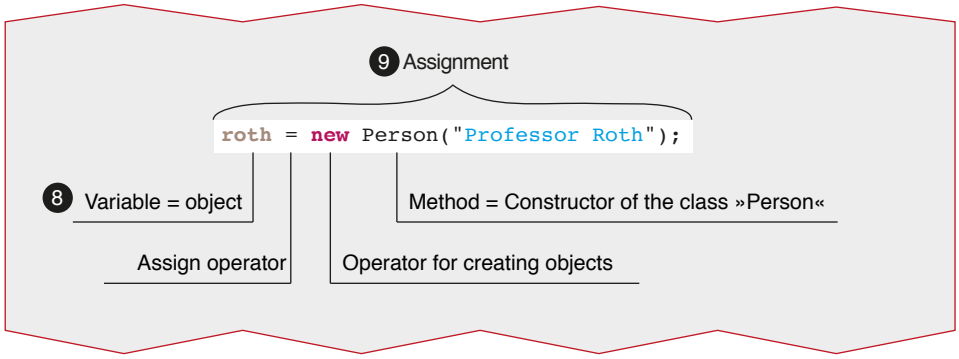


Figure 5.11: The assignment assigns a value to the variable.

this.name = name statement is beyond the scope of this chapter. The explanations of these statements are the focus of the chapter 7, »Statements«.

```
class Person {

  String name; // Attribute "name"

  public Person(String name) { // Constructor
    this.name = name;
  }

  public String getName() { // Getter method
    return name;
  }

}
```

Figure 5.12: The class »Person« and its constructor.

The method calls another method for text output at the end (Chapter 12, »Methods«). The class consists of a block that begins and ends with a curly bracket. The method *main()* also contains a block that begins with a curly bracket and also ends again. In addition, a number of keywords are marked in the text, which the 5.6 section of this chapter takes a closer look at.

5.4 Structure of the Program

Figure 5.13 again shows an overview of how the program *ProgramDemo* is structured. It consists of a class named *ProgramDemo* and a class named *Person*. Both classes belong to a package named *programmingcourse*. A package groups together one or more classes. In this case, the package groups together all the classes that belong to Professor Roth's programming course.

The class *ProgramDemo* contains the method *main* and can thus be executed as a Java program. Here, the *main* method is called from outside and takes control of the program flow. The method uses the class *Person* to create an object called *roth* and output its name. To understand the structure of the program, a structure diagram like the one in Figure 5.13 is the best way. On the other hand, to understand the flow of the program, it is best to use a flowchart.

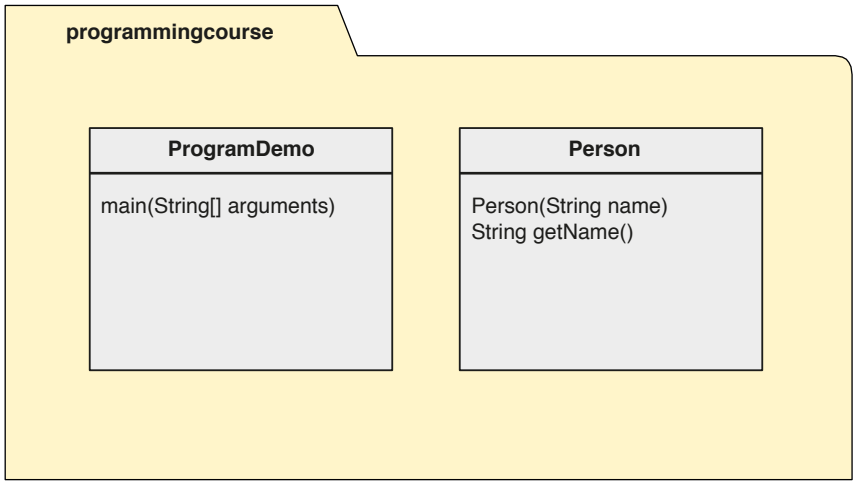


Figure 5.13: This structure diagram shows the structure of the »ProgramDemo« program.

5.5 Program Flow

Figure 5.14 represents the flow of the program. A user starts the program *ProgramDemo* from the outside via the method *main* (Point 1). A method is the object-oriented term for a function. This method creates an object called *roth* of type *Person* (Point 2). By »type Person« is meant that the object *roth* is an instance of the class *Person*.

The program assigns the string *Professor Roth* to the object when it is created. Finally, in the next line, the program prints the name of the professor on the screen. To do this, the program calls the method *println* of the *System* class (Point 3). This method in turn calls the method *getName* of the object *roth* (Point 4) and thus outputs the name of the professor.

So, in summary, the short program consists of a class *ProgramDemo* that creates an object called *roth* and gives it the name *Professor Roth*. Finally, the program prints the name of the object *roth* to the screen using the methods *println()* as well as *getName* and then ends.

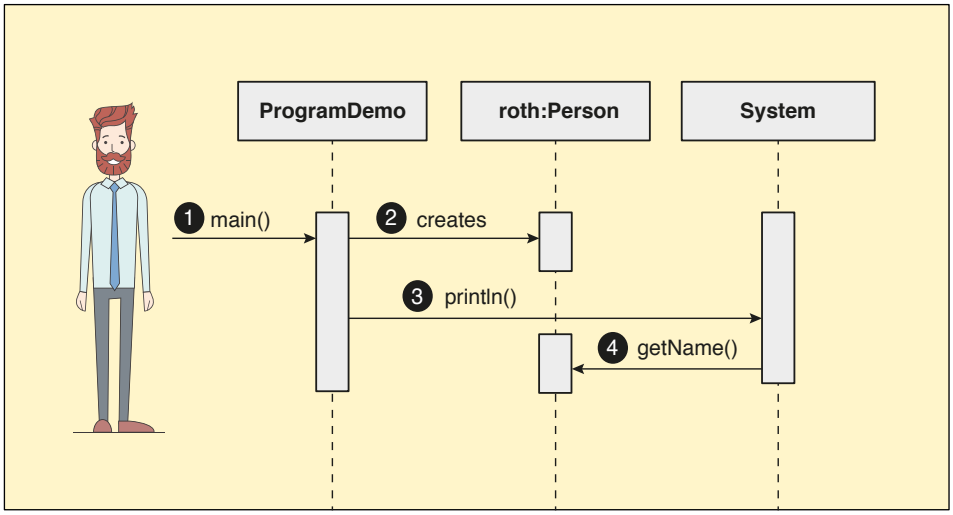


Figure 5.14: This flowchart shows the flow of the program »ProgramDemo«.

5.6 Reserved Keywords

When you take a closer look at the program example, you will notice some highlighted terms that have a reserved meaning in Java (Listing 5.2).

Listing 5.2 The program »ProgramDemo« as file »ProgramDemo.java«

```
1  //Examples/Program_Structure
2
3  package programmingcourse;
4
5  public class ProgramDemo {
6
7    public static void main(String[] args) {
8      Person roth;
9      roth = new Person("Professor Roth");
10     System.out.println("The person is called " + roth.getName());
11   }
12 }
```

These words are called keywords. They have a fixed meaning in Java. Therefore, you cannot name a class or variable as one of these Java keywords. According to the Java language definition, 57 of these fixed Java terms currently exist (Table 5.1).

The Java programming language has remained very stable since its inception. The developers of the programming language have added only a few keywords since Java first appeared: For example, the keyword *strictfp* has been part of the language since Java 1.2, *assert* was added in Java 1.4, while *enum* has only existed since Java version 5 (JDK 1.5). The keywords *exports*, *module* and *requires* were added in Java 9 (JDK 1.9), while the keyword *var* has only been assigned since Java 10.

Table 5.1 Reserved keywords

abstract	*assert*	*boolean*	*break*	*byte*
case	*catch*	*char*	*class*	*const*
continue	*default*	*do*	*double*	*else*
enum	*exports*	*extends*	*false*	*final*
finally	*float*	*for*	*goto*	*if*
implements	*import*	*instanceof*	*int*	*interface*
long	*module*	*native*	*new*	*null*
package	*private*	*protected*	*public*	*requires*
return	*short*	*static*	*strictfp*	*super*
switch	*synchronized*	*this*	*throw*	*throws*
transient	*true*	*try*	*var*	*void*
volatile	*while*			

There are two special keywords, *const* and *goto*. They have been reserved in Java, but may not be used. The keyword *const* may not be used in Java because Java prescribes a different syntax for constants. The case is different for the goto statement. It must not be used to avoid problems that such references can cause. The fact that the two keywords were reserved without allowing their use is due to the fact that they belong to the language scope of C/C++. Switchers who use these keywords should receive a meaningful error message.

Keywords such as *case*, *switch*, *break*, or *if* are reserved for branches. Branches allow the programmer to make case distinctions. Depending on the case, the program flow then branches into one or another program part. Recurring sequences, called loops, use the keywords *do*, *for* and *while*. Loops are necessary to read in files line by line, for example. Another use case is that the program should output data in the form of tables.

Most keywords are reserved for or in connection with data types. You already know the data type class. A class is identified by the keyword *class*. Classes like the *Person* class of this example can be defined by the programmer. Other data types, such as *boolean* or *int*, are hard-coded into Java. You can simply use them without having to define anything.

5.7 Summary

In this chapter, you have seen an example of how a simple Java program is constructed. It consists of the main class *ProgramDemo* and an object called *roth*. The program creates this object from the class *Person*. Classes are stored in text files with the same name. They consist of attributes (variables) and methods. All these components are just instructions to the program. These instructions determine how the program is executed.

To implement these instructions, Java has a whole set of reserved keywords with different meanings. For example, there is a keyword for defining classes such as the class *Person* or a keyword for creating objects using the new operator. Other keywords are reserved for conditionals such as *if*, others for loops such as *for*.

```
• Variables: e.g. roth
• Statements: e.g. roth = new Person();
• Classes: e.g. class Person ...
• Objects: e.g. roth
• Primitive data types: e.g. int
• Enumerations: e.g. enum
• Methods: e.g. main()
• Operators: e.g. new
• Conditional statements: e.g. if
• Loops: e.g. for
• Packages: e.g. package programmingcourse
• Exception handling: e.g. try
• Documentation: e.g. // Author K. Roth
```

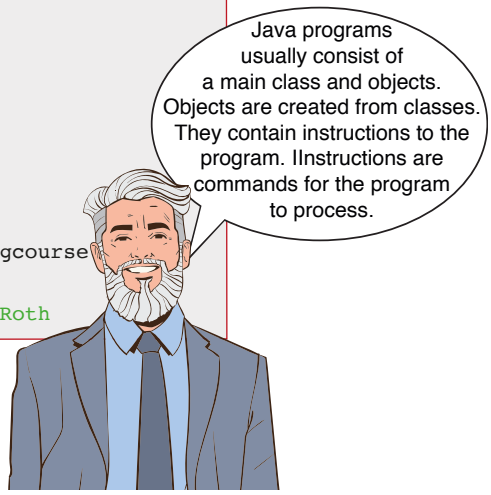


Figure 5.15: Professor Roth summarizes the structure of a Java program.

5.8 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/program_structure

5.9 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge:
https://www.programmingcourse.net/courses/java_with_eclipse/program_structure
There you will also find the solutions to the exercises.

6 Variables

6.1 Introduction

The previous chapter showed you how a Java program is basically structured using the example of the simple program. It consisted of a class called *ProgramDemo* with the method *main*. This method created an object using the class *Person*.

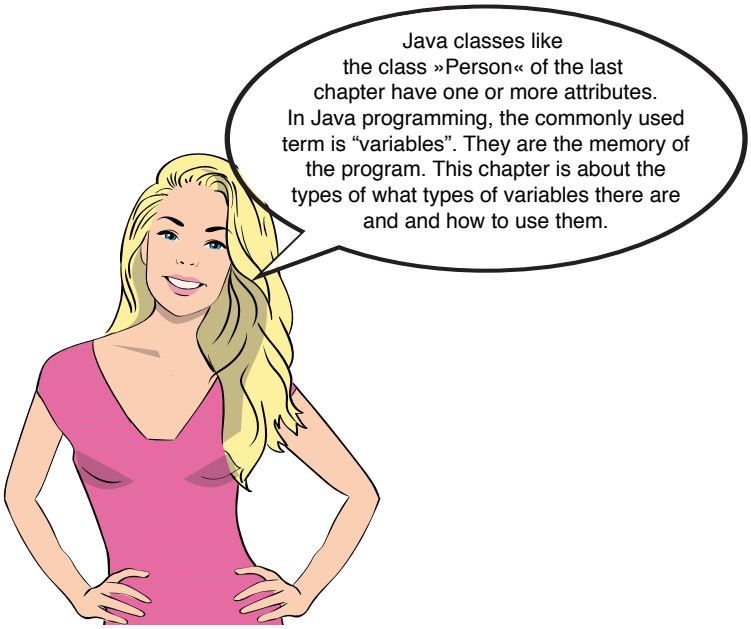


Figure 6.1: This chapter shows what types of variables there are and how to use them.

This chapter revisits this previous example. This time the focus is on the attributes of the class *Person*. In Java programming, attributes are usually called variables. Java provides several types of variables.

6.2 Overview

6.2.1 Variable Purpose

A Java program consists of objects. Objects are composed of methods, attributes and other objects. Java programs use variables to store attributes and objects. Variables are, so to speak, the memory of the objects of a program. To protect attributes from unauthorized access, an object surrounds them with a shell of methods (Figure 6.2).

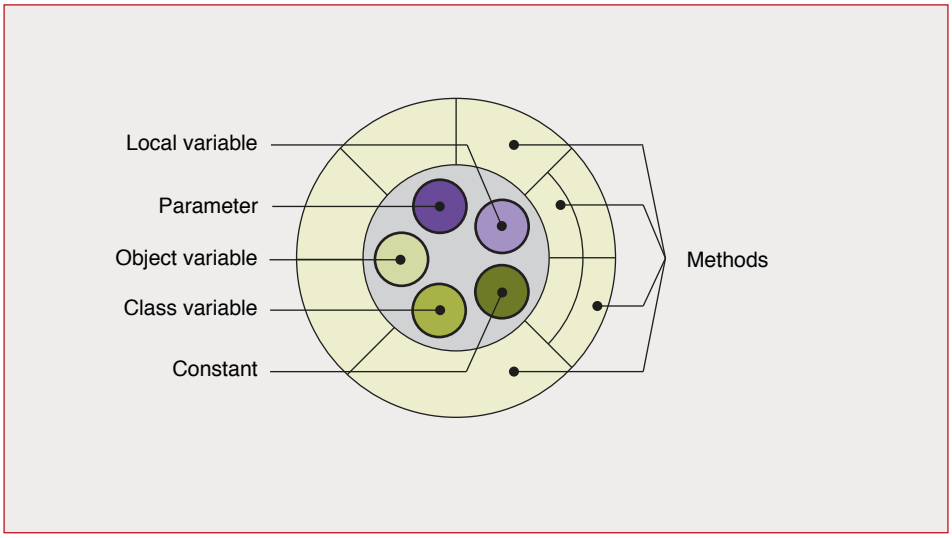


Figure 6.2 An object is similar to a cell with a protective capsule.

6.2.2 Variable Types

The first part of this book has called variables attributes of objects. Here there were properties like the height of a person and its name or states like the attribute *student*. Java maps the various attributes of the object-oriented world to five different types of variables (table 6.1).

Table 6.1 Overview of the types of variables.

Description	Type
Variable	Local variable
	Parameter
	Instance variable
	Class variable
Constant	Static variable (»variable« with unchangeable value)

6.2.3 Variable Usage

Each variable you want to use must be declared. Through the declaration, the program specifies which data type a variable should be. Java is a so-called typed programming language. The purpose of typing is to prevent errors when running programs. For this purpose, the Java compiler checks while it compiles a program, if all assignments of the program fit the declarations. For this reason, you must always specify the type of a variable you want to use. In Java, you must first specify the data type and then the name of the variable (Figure 6.3).

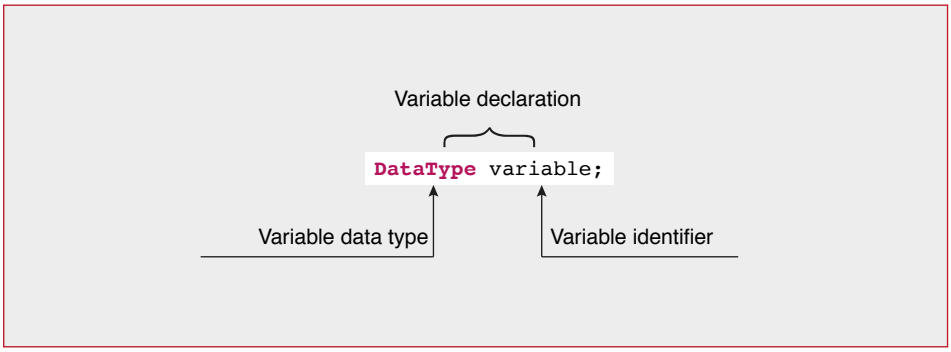


Figure 6.3: Declaration of a variable.

After the declaration, you can assign a value to the variable. Figure 6.4 shows how the assignment of a value is structured. The assignment always runs from right to left in a Java program. The direction of the course of an assignment means: On the left side is the variable, and on the right side is the value that is to be assigned to the variable. In Java, by the way, variables are always written in lower case.

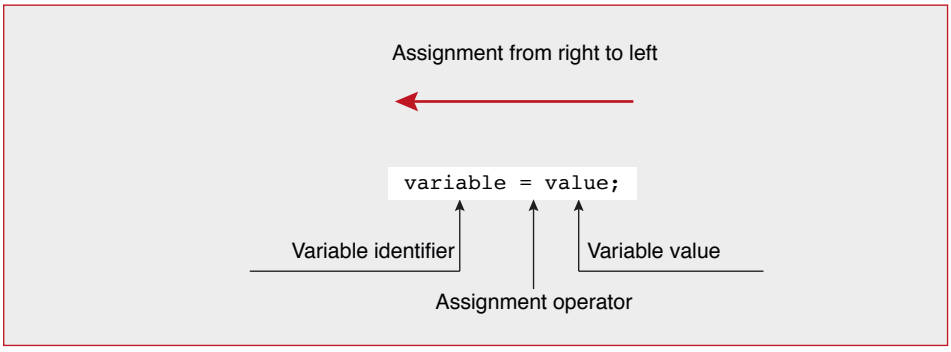


Figure 6.4: Assigning a value to a variable.

In many cases, the separation between declaration and assignment is necessary. In other cases it is cumbersome. For this reason, declaration and assignment can also be combined into a shorthand notation (Figure 6.5).

An example of a declaration with concrete values: A variable is to be assigned a height of 171 cm. To do this, it is first necessary to think of a name for the variable. The length of a

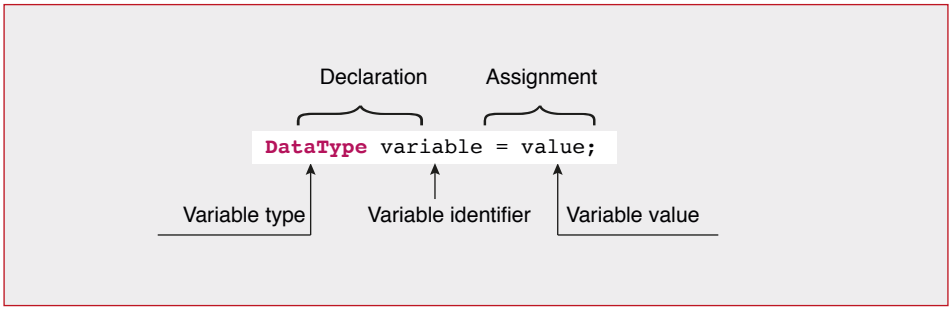


Figure 6.5: Combination of declaration and assignment.

variable name in Java is practically unlimited. Nevertheless, you should avoid long-winded variable names, as they make the program more difficult to read. A good variable name is as simple and self-explanatory as possible. You can use national or other special characters, but I do not recommend it. In the case of the example, the variable name is *height*.

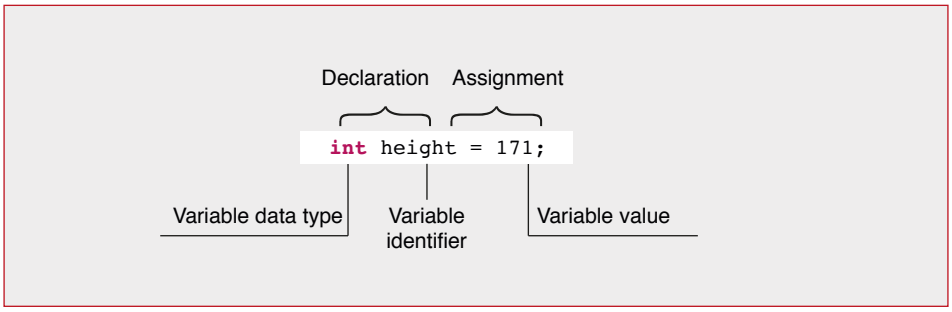


Figure 6.6: An example of a declaration and assignment to a variable.

After the name, the program must declare of which data type the variable *height* should be. To do this, consider which data type fits the variable *height*. In this case, it is the data type *int*. Data types like *int* will be introduced later in this book. Just so much as a foreshadowing for chapter 8, »Primitive Data Types«: The data type *int* allows to store integer values and is therefore suitable for the variable *height*. Once the declaration is complete, the program can assign the value *171* to the variable. Figure 6.6 shows what the combination of declaration and assignment looks like. As you can see, the unit of length has been dropped under the table. So for this simple program, you must remember that it works with the unit centimeter.

After the label, the program must declare what data type the variable *height* should be. To do this, consider which data type fits the variable *height*. In this case, it is the data type *int*. Data types like *int* will be introduced later in this book. Just as much as a preface to chapter 8, »Primitive Data Types«: The data type *int* allows to store integer values and is therefore suitable for the variable *height*. Once the declaration is complete, the program can assign the value *171* to the variable. Figure 6.6 shows what the combination of declaration and assignment looks like. As you can see, the unit of length has fallen under the table. So you have to remember for this simple program that it works with the unit centimeter.

6.3 Local Variables

Local variables have the task of storing data locally. Local means that the variable is only visible in a special section of a class and therefore not globally within the class. One always stores data locally only if a variable is exclusively needed locally. This can be, for example, if one receives an intermediate result of a calculation or if an object is only needed locally. The declaration of a local variable looks the same as for other variables. However, the visibility of the variable is limited only to the block in which it was declared (Figure 6.7).

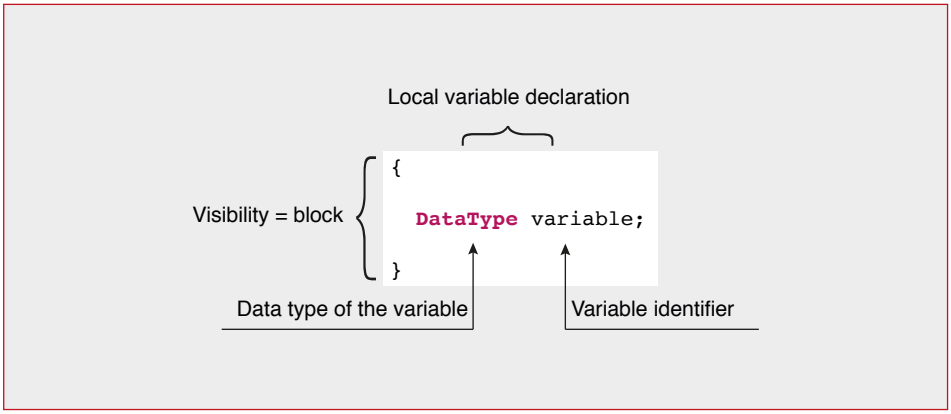


Figure 6.7: Declaration of a local variable.

An example of a local variable is the object *roth* from the class *LocalVariableDemo*. Have a look at the listing of the method *main* from the *LocalVariableDemo* class (Listing 6.1).

Listing 6.1 The local variable »roth« inside the method »main«.

```
7   public static void main(String[] arguments) {
8
9     Person roth;
10
11    roth = new Person("Karsten", "Roth");
```

On line 9 of the example, the program declares the variable *roth* of data type *person*. The program assigns it the object, which is created by the expression *roth = new Person("Karsten", "Roth")*. The life of the variable *roth* is limited to the method *main()*. Outside of the *main* method in the rest of the class, this variable is not visible. This also means that the object is perfectly protected from access by other objects in the program, because it has no interface to the outside world.

6.4 Parameters

Parameters are similar to local variables. Parameters are also only valid within a block. Unlike a local variable, however, the declaration is not inside a block. Instead, it is part of the so-called method signature. The method signature defines the interface of the class to which the method belongs. Figure 6.8 shows a method that contains a variable called *parameter*. This parameter comes into the method from the outside. For example, this parameter can be assigned to a local variable.

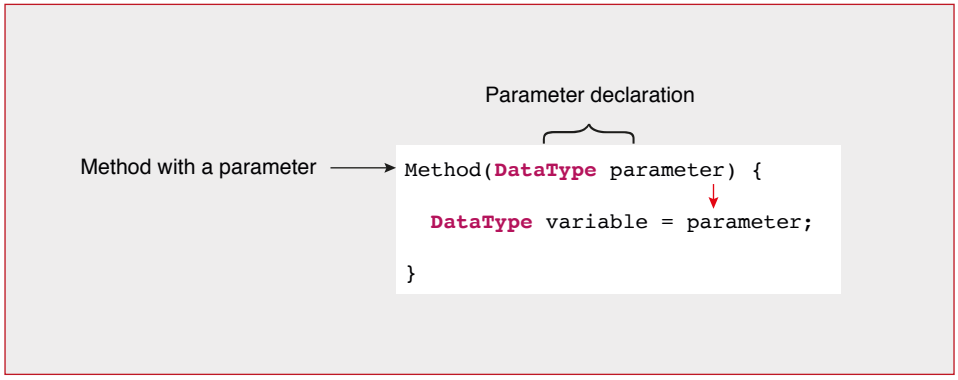


Figure 6.8: Declaration of a single parameter.

A program can take not only *one* parameter, but various ones. An example with the extended class *Person* shows this. Here the constructor of the class *Person* contains two parameters: the first name and the last name. Both declarations of the parameters must be separated by a comma. Once the two strings have entered the method, the program can then process them (Figure 6.9).

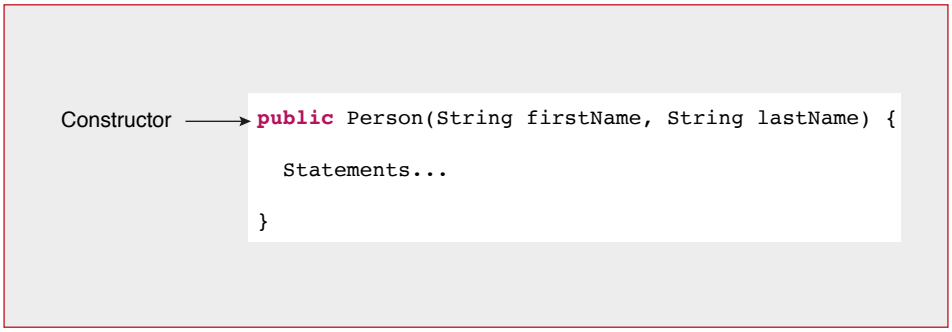


Figure 6.9: Example of the declaration of multiple parameters.

6.5 Instance Variables

In Java jargon, the natural attributes of a class are called instance variables. Natural attributes can be, for example, a first name or a last name - as in the example on the previous page. Instance variables are object-specific, hence their name. This means that each object is given individual values when it is created (Figure **??**).

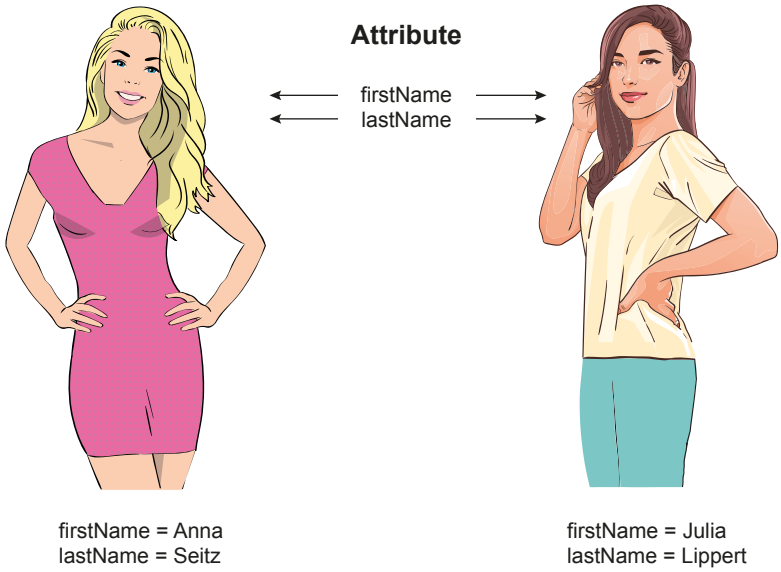


Figure 6.10: Instance variables are the individual attributes of the objects.

6.5.1 Individual Instance Variables

You declare individual instance variables right at the beginning of a class. They are thus visible within the entire class (Figure 6.11).

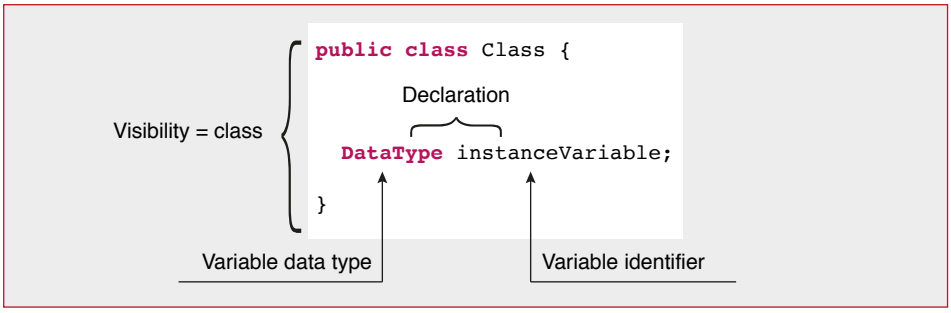


Figure 6.11: Declaration of an instance variable.

6.5.2 Instance Variable »this«

There is a special instance variable, *this*, which is automatically created by the compiler. It is used to always access the current object. It can be used in various situations. Most often, *this* is used to distinguish parameters and attributes of the same name. For this, again the example with the class *Person* (Listing 6.2). The class consists of an attribute *name* of type String and a constructor. The constructor has a parameter with also a variable of the same name and type.

Listing 6.2: The class »Person« with two attributes.

```
1  //Examples/Variables
2
3  package programmingcourse;
4
5  public class Person {
6
7    String name; // Attribute name
8
9    public Person(String name) {
10     this.name = name;
11   }
12
13   public String getName() {
14     return name;
15   }
16 }
```

The class *ThisDemo* calls the constructor of the class *Person* in line 8. Here, *ThisDemo* passes the name of the person to create an object called *roth* and named *Professor Roth* (Listing 6.3).

Listing 6.3: The class »ThisDemo« creates a new person object.

```
1  //Examples/Variables
2
3  package programmingcourse;
4
5  class ThisDemo {
6
7    public static void main(String[] arguments) {
8      Person roth = new Person("Professor Roth");
9      System.out.println("The person is called " + roth.getName());
10   }
11 }
```

For better understanding, the Figure 6.12 shows the whole process again in overview. To make it easier to follow the individual steps, they are numbered consecutively in the figure. The program *ThisDemo* calls the constructor of the class *Person* with the *new Person(...);* statement to create a new object of type *Person* (Point 1). In this call, the program passes the name of the person in the form of the parameter *name* (Point 2). The parameter *name*

has the effect of a local variable in the constructor of the class. It covers the attribute of the class with the same name within the method. Therefore, the program must access the *name* attribute of the class in the left part of the assignment (Point 4) with the *this.name* command.

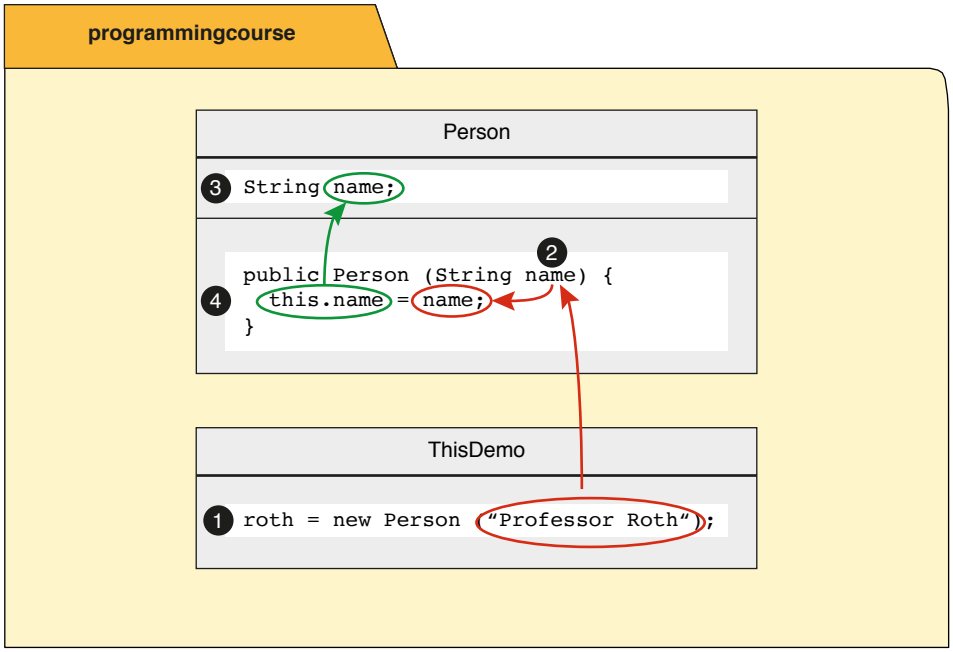


Figure 6.12: Usage of object variable »this«.

Of course, there is another way. In the example in Listing 6.4, the parameter distinguishing the class attribute is called *namePerson*. Since its name is different from the class attribute *name*, it does not obscure it locally. Accessing it using *this* is therefore unnecessary. But you can clearly see why *this* is generally used in professional programming. Without using *this* you always have to come up with another name for a similar variable.

Listing 6.4: Constructor without using »this«.

```
1 public Person(String namePerson) {
2   name = namePerson;
3 }
```

If Java is your first programming language, hiding a class attribute with a local variable will probably seem strange to you. At this point, however, I do not want to go into this topic any further. The third part of the book takes over and goes into these rules in detail in chapter 23, »Rules«.

6.6 Class Variables

While object variables are private property of objects, so to speak, it is different for class variables. Class variables apply to *every* object of a class in which they were declared. One declares them by the keyword *static*. As with the other variable types, declaration and assignment can also be combined (Figure 6.13).

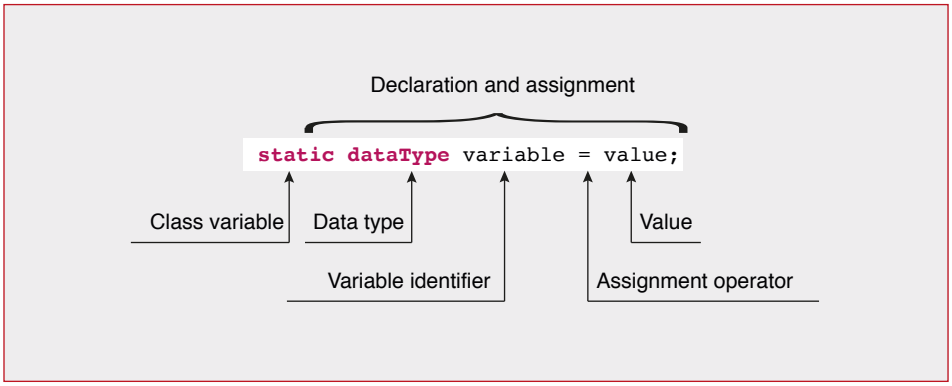


Figure 6.13: Class variables apply to any object of the class in which they are declared.

Class variables are therefore not bound to an object. Therefore, they exist from the time a class is loaded until the program terminates. In other words, every object created by the class has a variable with exactly the *same* value. The following example shows what consequences this has. The example consists of the class *person* and the program *class variable demo*. The class *Person* has the attribute *student*. The program initially assigns the initial value *false* to this variable. By calling the constructor of the class in line 9 of the listing, this initial value is assigned a suitable value for an instance of the type *Person* (Listing 6.5).

Listing 6.5: The class »Person« with the class variable »student«.

```
1  //Examples/Variables
2
3  package programmingcourse;
4
5  public class Person {
6
7    static boolean student = false;
8
9    public Person(boolean student) {
10     Person.student = student;
11   }
12
13   public static boolean isStudent() {
14     return student;
15   }
16 }
```

Of course, not every person is a student. For example, in Professor Roth's programming course, Anna is a student, but Robert is not. He is a robot. To show this, the following program creates two person objects with different values, as if they were object variables. The program first creates an object called *anna* on line 9 and asks whether she is a student on line 10. It then creates an object named *robert* in line 12 and asks whether Robert is a student in line 13 for this person object as well. As a check, the program asks again in line 15 whether Anna is a student (Listing 6.6).

Listing 6.6: The program calls the variable without an object.

```
1  //Examples/Variables
2
3  package programmingcourse;
4
5  class ClassVariableDemo {
6
7    public static void main(String[] arguments) {
8
9      Person anna = new Person(true);
10     System.out.println("Is Anna a student? " + anna.isStudent());
11
12     Person robert = new Person(false);
13     System.out.println("Is Robert a student? " + robert.isStudent());
14
15     System.out.println("Is Anna a student? " + anna.isStudent());
16   }
17 }
```

The output of the program is as follows:

```
Is Anna a student? true
Is Robert a student? false
Is Anna a student? false
```

The result is astounding, because after the last assignment in line 12, by creating a person object that is not a student, Anna has also lost her student status. The reason is that all copies of the class *Person* have the same *student* variable. If you change the value of this class variable for an object, you change all values of all copies that have been created so far. For this reason, the use of class variables should be considered carefully.

A word about the syntax of the program: If you load this example with Eclipse, you will notice that the development environment marks the statements *anna.isStudent()* and *robert.isStudent()* with warnings. Why? The syntax of the statements is formally correct. However, the spelling gives the impression that the programmer wants to access an object variable. To access static variables, use a different notation for the query. With this, the query is as follows:

```
Person.isStudent();
```

By using this type of query, the class is named first and then the method, rather than the object and then the method. This makes it immediately clear to any well-trained Java programmer that the program is querying a class variable here, not an object variable. With an

object variable a query of this kind would not function. Since all class variables always have the same value, a query like in lines 10, 13 and 15 of the Listing 6.6 is therefore completely misleading. Therefore, avoid object variable style queries of class variables.

6.7 Constants

From class variables, it's only a short jump to constants. But does Java support constants? The way to a constant seems to be quite obstructed: The keyword *const* is reserved, but may not be used. If you want to define constant values, you mark them in the variable declaration by the keywords *static final*.

The keyword *static* declares a class variable. The *final* keyword makes its value immutable. It may be assigned only once in the program. So if you want, this kind of constant is an enhancement of the class variable presented before. Usually, the introduction of a constant happens in a combination of declaration and assignment, as you already know it with other variables (Figure 6.14).

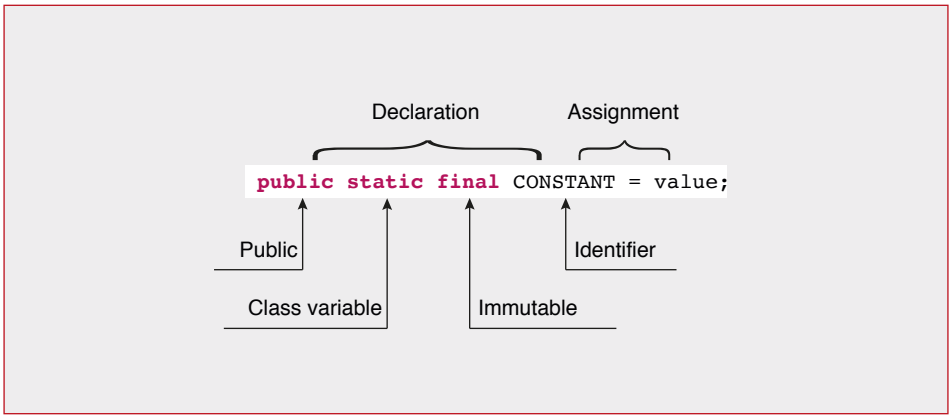


Figure 6.14: Constants are immutable class variables.

If you want to create a constant, you combine the keyword *static* with *final*. Both are preceded by the keyword *public*. So the constant is accessible to everyone even without a query method. This also makes sense, since encapsulation is pointless, because no one can change the value of a constant anyway.

Let's return to Professor Roth's programming course. The following example shows how a constant works for the class *Person*. The class has only one attribute, the number of legs. The number is set to the correct value of 2 right at the beginning and written in uppercase (Listing 6.7). By writing the constant in capital letters throughout, it is immediately clear when reading the program at any point that it is an unchanging value.

Listing 6.7: The program has the constant attribute »NUMBER_OF_LEGS«.

```
1 //Examples/Variables
```

```
2
3  package programmingcourse;
4
5  public class Person {
6
7    public static final int NUMBER_OF_LEGS = 2;
8
9  }
```

The program *ConstantDemo* does not need to create an object to access a constant (Listing 6.8). This is unusual at first and seems to contradict the essence of an object-oriented program. A constant is created with a class and leads, so to speak, an existence of its own without an object of a class.

Listing 6.8: The program calls the variable without an object.

```
1  //Examples/Variables
2
3  package programmingcourse;
4
5  class ConstantDemo {
6
7    public static void main(String[] arguments) {
8
9      System.out.println("Persons have " + Person.NUMBER_OF_LEGS +
10       " legs");
11
12   }
13 }
```

A query method is unnecessary for constants, since constants are generally declared as *public*, as in the class *Person*. Why not, since no one can change the value of the constant? So any part of the program can access this immutable »variable« unhindered. Consequently, a query method is superfluous for public attributes. When you start the program *ConstantDemo*, it prints the following information on the console:

```
Persons have 2 legs
```

6.8 Summary

In this chapter, you have learned which variables and constants can be used in Java. Java distinguishes four types of variables besides the constant: local variables, parameters, and object and class variables. Local variables apply only where they have been declared. They are therefore locally limited. Parameters, similar to local variables, apply only in the block of a method. However, they are declared in the method header. Object variables are different for each object (copy) of a class. In contrast, class variables are static, as specified when they are declared. Once set, the value of a class variable is the same for all copies of a class.

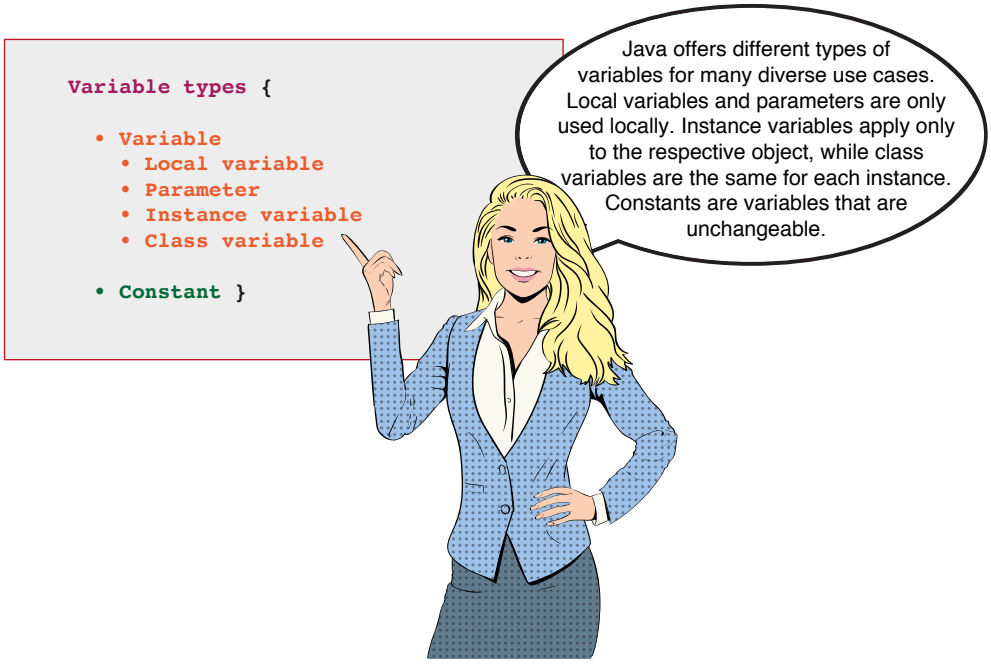


Figure 6.15: Java provides several different types of variables as well as constants.

Constants go one step further. They have a special role in Java, because they are nothing else than class variables with immutable value. The keyword *static* specifies that they are class variables, and the keyword *final* that they are immutable. To emphasize constants, they are written in capital letters. By declaring them as *public*, unlike other attributes, you don't need query methods to access their value.

6.9 Literature

Type signature: *https://en.wikipedia.org/wiki/Type_signature*
Variables: *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html*

6.10 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/variables

6.11 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge: *https://www.programmingcourse.net/courses/java_with_eclipse/variables*

There you will also find the solutions to the exercises.

7 Statements

7.1 Introduction

This chapter discusses statements before the remainder of the book presents Java programming from the perspective of classes and objects. Statements are elementary components of every Java program. You can compare them very well with sentences or commands of a natural language. From a somewhat more technical point of view, a statement in the sense of programming is a »complete execution unit«. The following examples show you what this means.

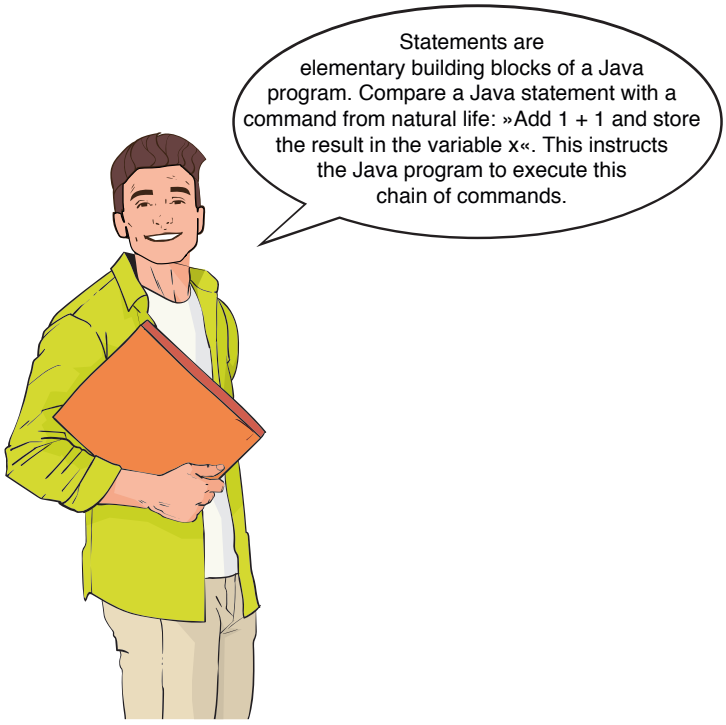


Figure 7.1: Florian hates statements. But they have to be in Java programs.

7.2 Overview

7.2.1 Statement Purpose

Java belongs to the imperative programming languages. Programs in these languages are composed of individual commands. In this type of traditional programming, the focus is not on *describing the problem*, but on *solving it*. The Java programmer is responsible for this solution. To do this, a coder enters the solution of the problem as a series of small commands. The program processes these commands exactly, piece by piece. This type of programming is called imperative (Latin imperare = »to command«). This leads to relatively long programs, but you can specify very precisely how the program works.

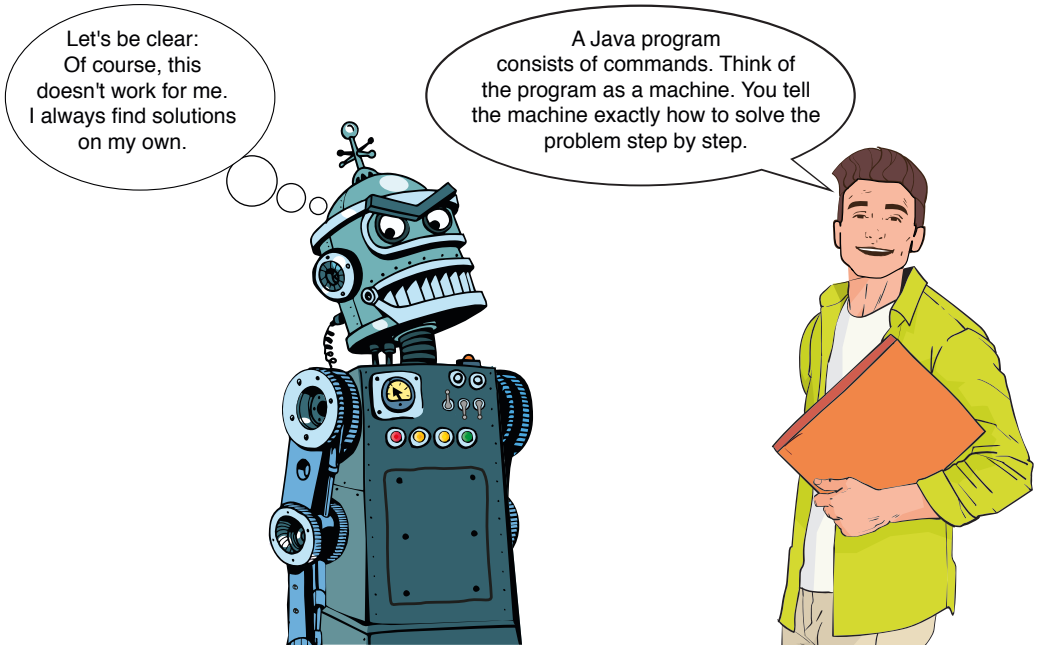


Figure 7.2: Robert believes that he can only be programmed declaratively.

There are programming languages where you proceed completely differently. In these programming languages, you formulate the problem and leave it up to the program how to solve the task. This type of programming is called declarative (Section 7.9, »Literature«). Declarative programming languages are, for example, the database query language SQL (Structured Query Language). With this language, you explain to the database system which data set must be found. The program then internally processes the functions that are necessary to find the data set. Since you do not know how the database system is structured internally, you leave it to the program to find the data on its own. This usually leads to amazingly short programs. However, you cannot determine how the program runs internally.

7.2.2 Statement Types

As mentioned at the beginning, you can compare Java statements with commands in the form of a sentence. What exactly does this look like in a Java program? Table 7.1 gives you some examples of different statements.

Table 7.1: Examples of Java statements.

Statement	Java Example
Declaration	String name;
Assignment	name = "Anna" ;
Variable call	Person.name;
Methode call	roth.getName());
Conditional statement	if (height > 171) ...
Loop	for (student = 1; i < 10); student++) {...}

7.3 Declaration

You must declare constants and variables in Java before you can use them. The structure of this instruction has already been briefly mentioned in the previous two chapters. How the declaration is structured is shown again in Figure 7.3. First comes the data type and then the constant or variable. The declaration is always part of a class, for example in the form of an *attribute*, a parameter or a local variable. This means you can never use an attribute like *name* detached from a class like *person*.

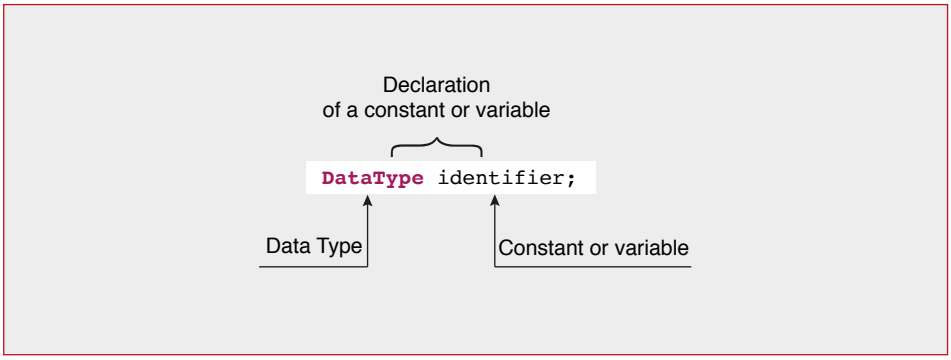


Figure 7.3: General structure of a Java declaration.

You can assign a value to the constant or variable only after the declaration. Declaring a constant or variable is necessary so that it is clear to the Java compiler how much memory it should allocate to the constant or variable. In addition, the declaration enables the compiler to control what may be done within the program with this constant or variable.

The declaration thus shapes two properties of the variable unchangeably for the lifetime of the program: Value range and arithmetic operations. Once a data type has been defined, it is unchangeable for the life of the program. The arithmetic operations that are bound to a variable are also unchangeable. For example, a string like *firstName* cannot be multiplied by another string like *lastName* (Figure 7.4). If the programmer had declared two numerical values instead, the program could perform the multiplication.

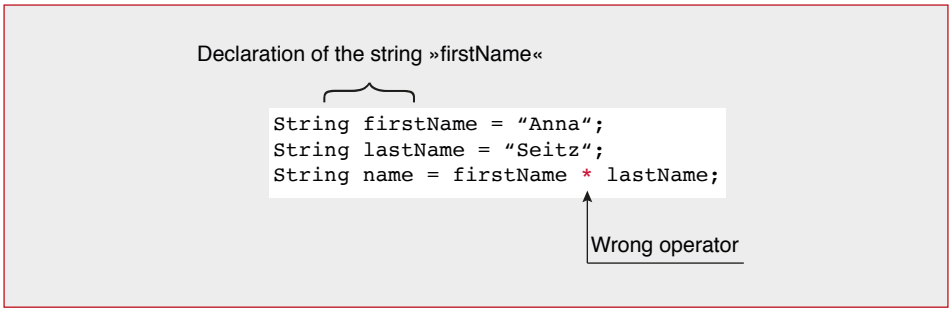


Figure 7.4: The multiplication of two strings is not allowed.

The declaration is concluded with a semicolon. This semicolon is very important. If it is missing, the program »runs« into the next line of source code. This continues until the program encounters a semicolon again. The end of such an instruction is therefore not the end of the line for the program, but the semicolon. Here again, the parallels between an instruction and a sentence in a natural language become clear. In a sentence, too, it is necessary to mark the end of the sentence, because otherwise the text would be difficult to understand.

Let's look at the whole thing again with a concrete example (Figure 7.5). On the left, the program declares a variable as a string with the Java class *String*, and to the right, the associated variable with the name *name*. This means that from this point on, it is specified that the variable *name* is a character string of the type *String*.

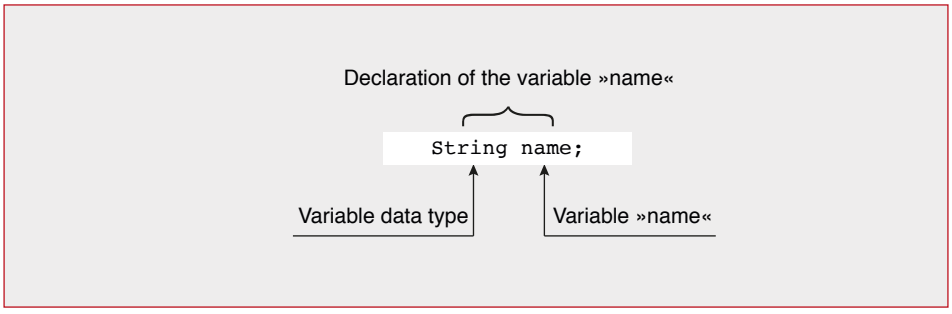


Figure 7.5: Declaration of variable »name«.

7.4 Assignment

7.4.1 Java Assignment Structure

The next step on the way to a working program is to assign a value to a declared variable. This type of statement is accordingly called an assignment. Assignments have been used implicitly several times in this book without explaining the process. The assignment looks like a mathematical equation (Figure 7.6).

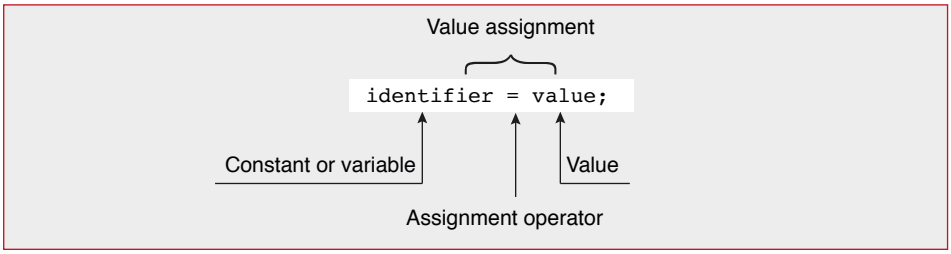


Figure 7.6: General structure of an assignment.

The equal sign is called an assignment operator in Java. It has a comparable effect to a method. For the computer, the sign is nothing more than the shorthand notation of a function. In this case, this function causes the value named *value* to be assigned to the memory cell named *name*. Hence the name »assignment«.

7.4.2 Java Assignments Are Not Equal to Mathematical Equations

Java assignments appear simple and therefore tend to cause misunderstandings among newcomers to Java programming. On the left side of the assignment are always the *variables* parts of the assignment. On the right side of the assignment are always the *fixed* parts of the assignment. The direction in which the program is processed is therefore exactly the opposite of the western reading direction, from right to left (Figure 7.7).

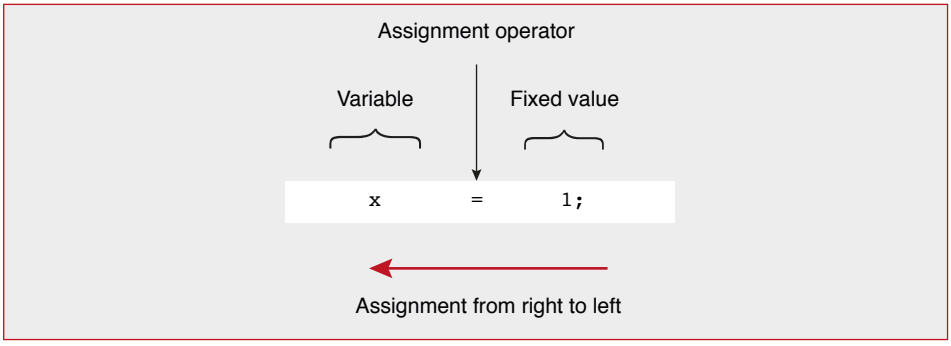


Figure 7.7: In Java, the assignment operator is an equal sign.

The direction in which a computer processes the assignment is not the only paradox. For example, an assignment like $x = 1$ appears to any Java programming novice as a mathematical equation. This misconception is caused by the fact that the Java programming language clumsily uses the equals sign, familiar from mathematics, as an assignment operator.

7.4.3 Is x = y Equal to y = x?

For comparison: An assignment in the programming language Pascal is very similar to the one in Java. Here, too, an assignment is processed from right to left. But Pascal uses a combination of colon and equal sign as assignment operator. An instruction thus looks like this:

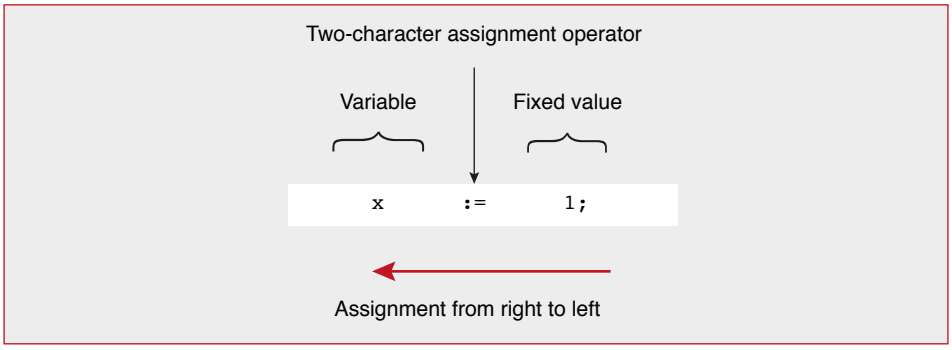


Figure 7.8: In Pascal, the assignment operator is different from the equal sign.

By this particular operator, Pascal emphasizes that the assignment operator works differently from the mathematical equals sign. The idea was to prevent anyone from thinking that $1 = x$ means the same as $x = 1$. In Java, a statement like $1 = x$ is not allowed, because on the left side there must be variables, that is, variable identifiers.

But what happens if there are variables on both sides? Is $x = y$ the same as $y = x$? In mathematics, definitely. In Java not at all. In the first case, the program assigns the value of the variable y to the memory cell x. In the second case, it is the other way round: the memory cell y gets the value of x. The program example in Listing 7.1 shows this clearly.

Listing 7.1: The expression x = y is not at all equal to y = x.

```
1  public class AssignmentDemo {
2
3    public static void main(String[] arguments) {
4
5      int x; // Declaration x
6      int y; // Declaration y
7
8      // Case 1:
9      x = 1;
10     y = 5;
11
```

```
12      x = y; // x receives the value of y
13
14      System.out.println("Case 1:\nx = " + x + ";\ny = " + y);
15
16      // Case 2:
17      x = 1;
18      y = 5;
19
20      y = x; // y receives the value of x
21
22      System.out.println("\nCase 2:\nx = " + x + ";\ny = " + y);
23    }
24  }
```

The program outputs the following:

```
Case 1:
x = 5;
y = 5

Case 2:
x = 1;
y = 1
```

In case 1, the computer has copied the value of the memory cell *y* to the memory cell *x*. In case 2, however, the memory cell y has received the value of the memory cell x. The Java programming language behaves differently from the mathematical language. Mathematically, it does not follow from x = 1 and y = 5 that x = y.

The quintessence of this example shows that mathematical formulas can by no means be transferred 1:1 into the Java programming language. You must therefore observe a number of rules, which chapter 23, »Rules«, will introduce in more detail.

7.4.4 Combination of Declaration and Value Assignment

As with objects, it is possible to combine declaration and assignment of a value for constants and variables. In cases where you need a local variable, this is simply more practical. This looks like Figure 7.9.

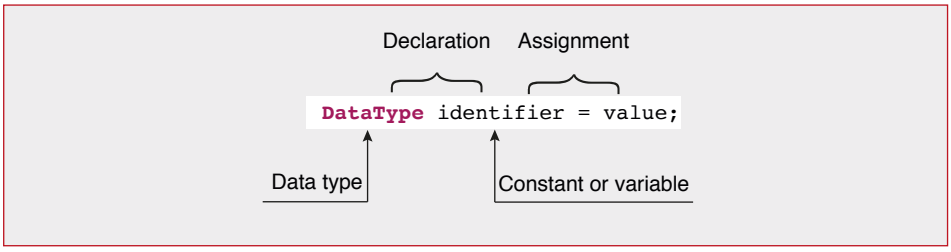


Figure 7.9: General structure of a declaration and an assignment.

Again, an example with a concrete value (Figure 7.10). The example shows the declaration of the variable *x* as an integer value. Integer is a Java data type for whole numbers.

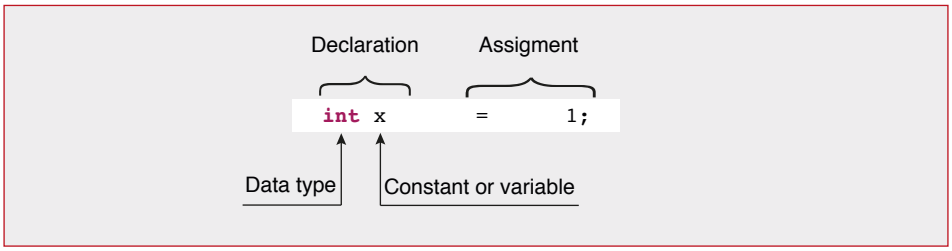


Figure 7.10: Example of a combination of declaration and assignment.

7.5 Block

A block group a series of logically related statements into a single unit. Each block begins with an opening curly bracket and ends with a closing curly bracket. The program executes the instructions within a block one after the other. A block can be, for example, the beginning of a class definition or that of a method. A Java program uses blocks mainly in classes, methods, branches and loops. They can be nested in each other in any way, as the example of a class with a method shows (Figure 7.11).

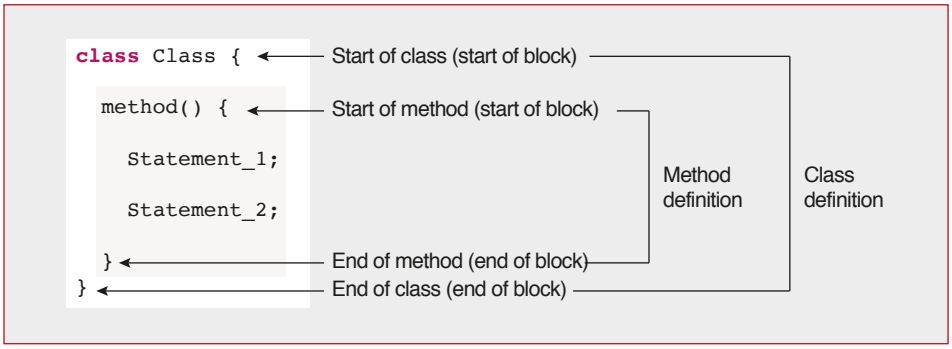


Figure 7.11: Formatting of a block in the style of the C programming language.

In this example, the opening curly bracket still starts on the line where the class and method definition starts. This very space-saving notation comes from the world of C programming and is widely used in Java. In my experience, this C notation seems somewhat crowded and therefore confusing to programming novices. If you do not like this notation, you can also use the notation of a block, as is common in the programming languages Algol and Pascal. Here, the block always begins in a separate line. This looks like in Figure 7.12. Eclipse has a code formatter that can format programs in any way. You can also set in the development environment how Eclipse should create classes, methods, conditional statements and loops.

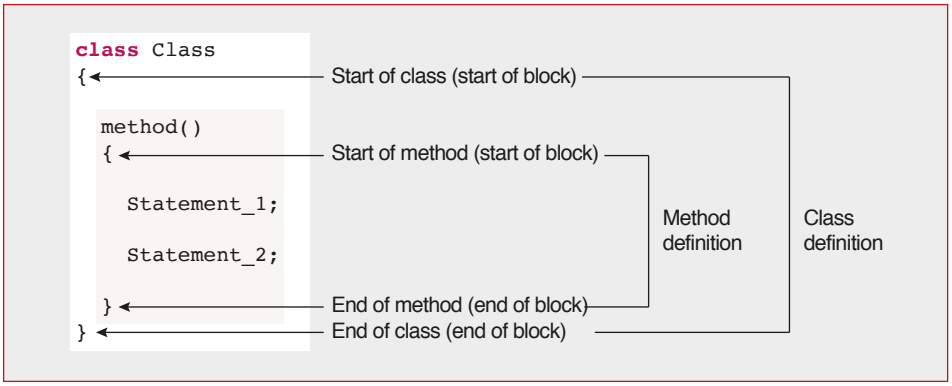


Figure 7.12: Formatting of a block in the style of the Pascal programming language.

You already know that a Java program always processes assignments from right to left. Within a block, each instruction is processed from top to bottom. Since the main class of a Java program consists of a block, this means that a Java program is also processed from top to bottom (Figure 7.13).

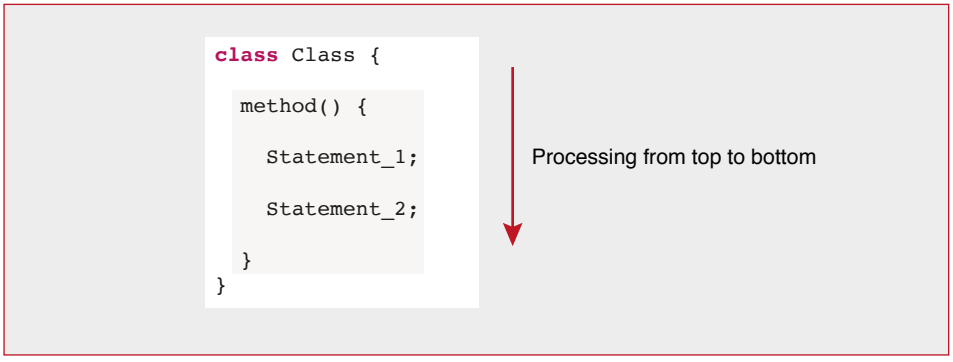


Figure 7.13: Processing of a block is done from top to bottom.

Blocks can have local variables that lose their validity outside the block. The following example declares a class variable called *direction* in line 7 and assigns it the value *West*. A block begins in line 10 and ends in line 13. In line 11, the program declares a variable called *group* and also assigns it a value. Line 12 outputs the two words as a unit. In line 17, this no longer works. Here, the variable *group* is no longer valid (Listing 7.2).

Listing 7.2: Blocks determine the validity of local variables.

```
1  //Examples/Statements
2
3  package programmingcourse;
4
5  public class WesternBlockDemo {
6
```

```
7    static String direction = "Western";
8
9    public static void main(String[] arguments) {
10     {
11       String group = "Block";
12       System.out.println(direction + "␣" + group);
13     }
14     System.out.println(direction + "␣" + group); // Incorrect statement!
15   }
16 }
```

The next example shows again how the validity of global and local variables is changed by a block (Listing 7.3). The following example declares a class variable named *direction* in line 7 and assigns it the value *Western*. Within the method *main*, you will also find a declaration of a local variable called *direction* with the value *Eastern* assigned to it in line 10. What happens? Which of the two variables wins?

Listing 7.3: The variable »direction« in line 7 is overlaid in the block.

```
1  //Examples/Statements
2
3  package programmingcourse;
4
5  public class EasternBlockDemo {
6
7    static String direction = "Western";
8
9    public static void main(String[] arguments) {
10     String direction = "Eastern";
11     String group = "Block";
12     System.out.println(direction + "␣"+ group);
13   }
14 }
```

The program output is as follows:

```
Eastern Block
```

So blocks give preference to the »weaker« local variables. This has to be the case because otherwise a local variable could no longer be accessed. However, the global variable has not disappeared, it is only hidden. By specifying in line 13 of the following example that it wants to access the class variable *direction*, the program also reappears in the program (Listing 7.4).

Listing 7.4: The variable »direction« in line 7 is overlaid in the block.

```
1  //Examples/Statements
2
3  package programmingcourse;
4
5  public class EasternWesternBlockDemo {
6
```

```
7    static String direction = "Western";
8
9    public static void main(String[] arguments) {
10     String direction = "Eastern";
11     String group = "Block";
12     System.out.println(direction + "␣" + group);
13     System.out.println(EasternWesternBlockDemo.direction + "␣" + group);
14   }
15 }
```

This time the program outputs the following:

```
Eastern Block
Western Block
```

The example shows that the variable *direction* in the block of the class *EasternWesternBlockDemo* can be accessed at any time in the inner block by an appropriate statement. The example probably looks familiar to you from the last chapter. There, a parameter in a block concealed a class attribute. It is the same principle of shadowing variables in a block (Chapter 23, »Rules«, Section 23.4.7, »Scope of Variables«).

7.6 Variable Call

Up to now, you have mainly encountered variables as attributes of a class. As a rule, you access these protected attributes indirectly via methods. In the case of the class variable *direction* of the previous example, you have seen that direct access can also be useful. Such a direct access is structured in two parts (Figure 7.14).

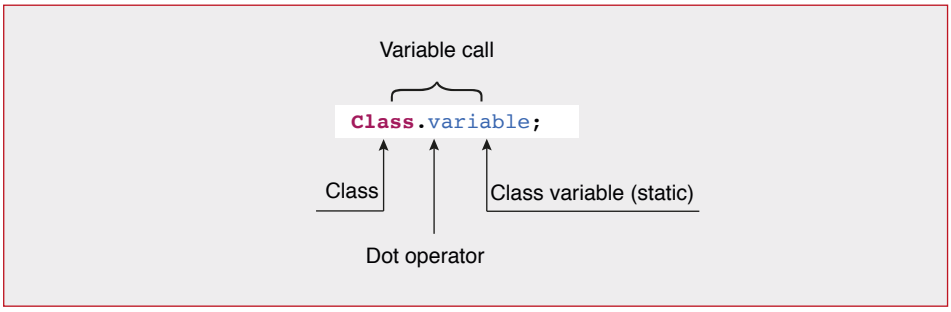


Figure 7.14: The variable call is divided into two parts.

The following example shows a program output that seems incomprehensible to most Java novices from my experience (Figure 7.15). On the left you see the class *system*. Using the dot operator, the program accesses the constant called *out*, which is part of the class *System*. The object *out* has been derived from the class *PrintStream*. Among other things, this class has the method *println*. It is able to perform a program output. The purpose of this cascade of instructions is to output something via the method *println*.

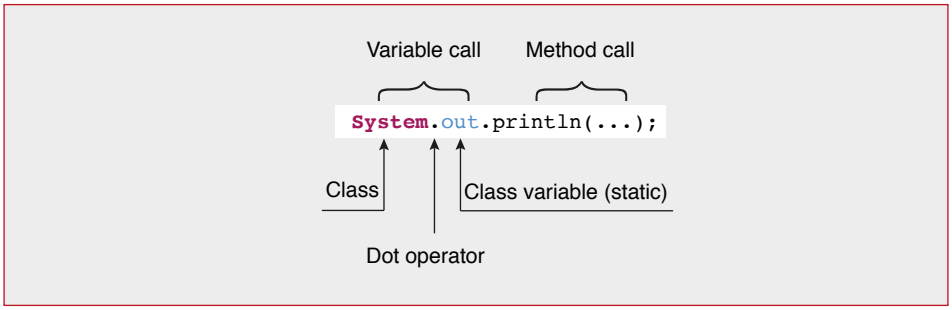


Figure 7.15: Example of calling the constant »out«.

7.7 Method Call

To make the complicated nested statement easier to understand, we will come to the method call at the end of this chapter. The call of a method, like the call of a variable, is always divided into two parts. On the left side is the object or class to which the method belongs. On the right side is the method. In the middle between the two must be the point operator for access (Figure 7.16).

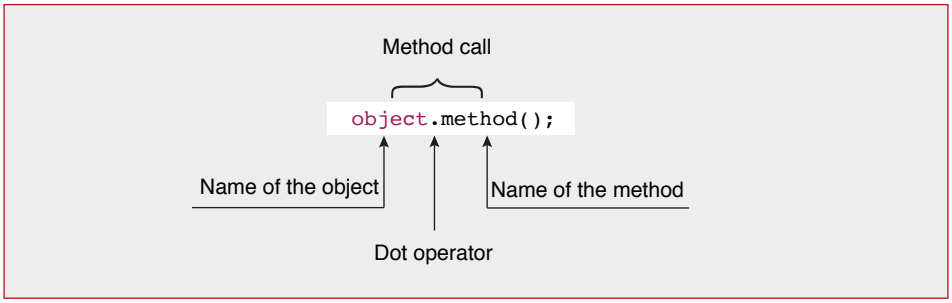


Figure 7.16: The method call is divided into two parts.

As can be seen, the method call can only be distinguished from a variable call by the pair of parentheses that methods generally have in Java. As an example of a method call, let's return again to the program called *ProgramDemo* (Listing 7.5). The program outputs the name of professor Roth in line 13 via the method *println*. To get the name of the professor, the program calls the method *getName* of the object *roth*.

Listing 7.5: The program »ProgramDemo« as an example of a method call

```
1  //Examples/Statements
2
3  package programmingcourse;
4
5  class ProgramDemo {
6
7    public static void main(String[] arguments) {
```

```
8
9       Person roth;
10
11      roth = new Person("Professor Roth");
12
13      System.out.println("The person's name is " + roth.getName());
14
15    }
16  }
```

7.8 Summary

This chapter has given you an overview of statements in Java programs. The declaration is one of the most important statements. It determines the memory location for a variable and which operators can be used with which variables. The assignment uses the previously declared variables to assign values to them. Here it is important that the program always assigns from right to left. On the left side are the changed values, and on the right the unchanged values. The chapter ended by introducing you to variable and method calls. At this point, the dot operator was introduced for the first time, which is used to access variables and methods.

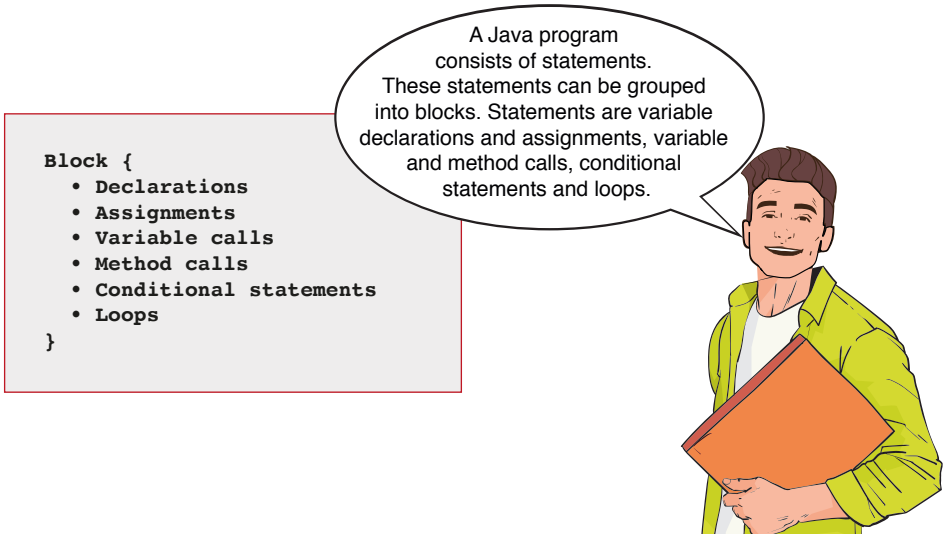


Figure 7.17: Florian now has a completely different attitude towards statements.

If you look at the enumeration of different statements in figure 7.17, you will notice that this chapter did not cover the statement types »Conditional Statements« and »Loops«. These types of statements are so important and extensive that they each got their own chapter. You will find »Conditional Statements« in chapter 14 and »Loops« in chapter 15.

7.9 Literature

»Declarative Programming«: *https://en.wikipedia.org/wiki/Declarative_programming*
»Imperative Programming«: *https://en.wikipedia.org/wiki/Imperative_programming*

7.10 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/statements

7.11 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge:
https://www.programmingcourse.net/courses/java_with_eclipse/statements

There you will also find the solutions to the exercises.

8 Primitive Data Types

8.1 Introduction

»Primitive« Java data types are used wherever pure numerical values or states have to be stored. With their help, objects store their attributes and exchange data. Primitive data types are not classes. Therefore, variables based on these data types are not objects. This seems to contradict the idea of object orientation. So why did the inventors of Java decide to use such data types? Robert has an answer:

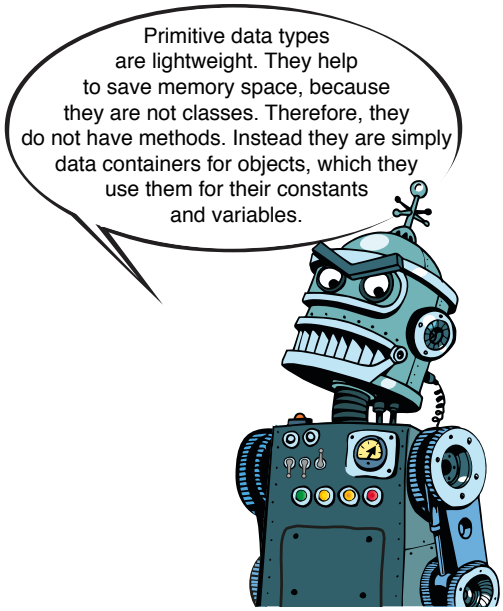


Figure 8.1: Primitive data types are lightweight, but not classes.

8.2 Overview

8.2.1 Purpose of Primitive Data Types

The goal of primitive data types was to limit the memory hunger of object-oriented programs. Since the primitive Java data types are no classes, they need only little memory. They used them everywhere, where an object with its methods is not necessary. This is particularly the case with simple attributes of classes, for example, a pure numerical value such as *height*.

8.2.2 Types of Primitive Data Types

In Table 8.1 you can see an overview of all primitive Java data types. They differ in their usage, how much memory they need and the value range. The value range determines the size of the reserved memory for a variable of this type.

Table 8.1: Overview of primitive Java data types.

Name	Type	Size	Value range	Default	Example
byte	integer	8 bit	-2^7 ... 2^7-1	0	1
short	integer	16 bit	-2^{15} ... $2^{15}-1$	0	2
int	integer	32 bit	-2^{31} ... $2^{31}-1$	0	3
long	integer	64 bit	-2^{63} ... $2^{63}-1$	0	5
float	decimal	32 bit	$\pm$ 3.40282347 * 10^{38}	0,0	8.13
double	decimal	64 bit	$\pm$ 1.79769313486231570 * 10^{308}	0,0	21,34
char	character	16 bit	All Unicode characters	\u0000	F
boolean	boolean value	8 bit	true, false	true	

There are the integer data types *byte*, *short*, *int* and *long*, the decimal data types *float* and *double*. Use the *char* data type to store single characters, and the *boolean* data type to store simple states with two values *true* and *false*.

8.2.3 Use of Primitive Data Types

Declaration

Primitive data types are part of the Java language. Therefore, unlike custom classes, they do not need to be defined, but only declared when used. The declaration is again identical to the variable declaration you already know from the chapter 6, »Variables«. Here you have to »explain« to the compiler of which simple data type a variable should be. In the declaration, you first specify the type of the variable and then the name of the variable. Figure 8.2 shows this using the example of the data type *int*.

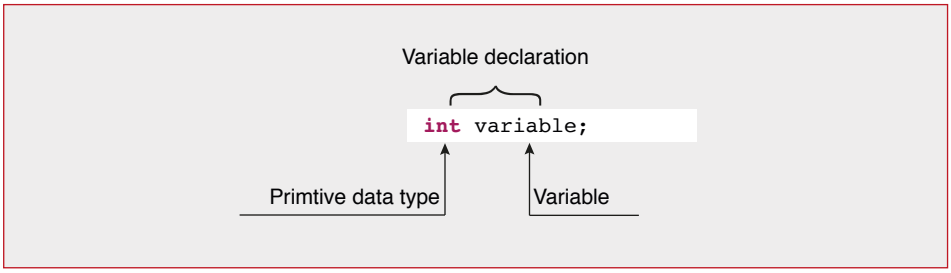


Figure 8.2: Declaration of a variable of a primitive data type using »int« as an example.

Assignment of a Value

After the declaration, you can assign a value to the variable. Using the example of a variable of value *int*, this looks like in Figure 8.3.

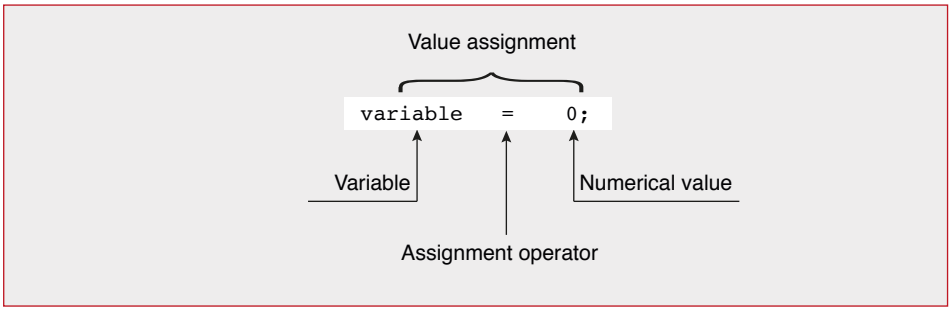


Figure 8.3: Assignment for a variable of a simple data type using the example of »int«.

Combination of Declaration and Assignment

As with all other data types, you can of course combine declaration and assignment if necessary. Using the example of a variable of value *int*, this looks like in Figure 8.4.

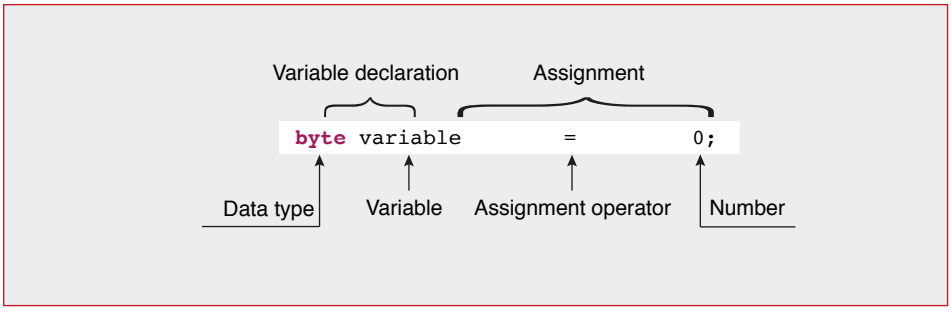


Figure 8.4: Declaration and assignment of a simple data type using the example of »int«.

Precision and Value Range

Primitive data types differ in the amount of memory they require. For example, a memory area of 4 bytes (= 32 bits) is reserved for the data type *int*. The reserved memory area is identical on all computer systems running a Java program. This is one of the great advantages of Java and one of the reasons why Java programs can be transferred relatively easily from one computer system to another.

The reserved memory area is not identical to the value area for numerical data types. This is because all Java numeric data types have a sign that requires one bit of memory. This means that, for example, the *int* data type has »only« 31 bits available (Figure 8.5).

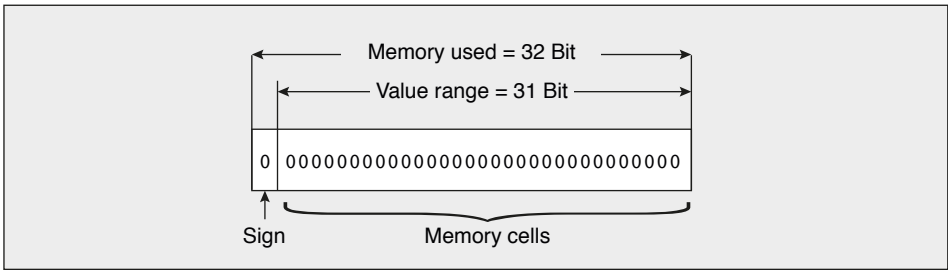


Figure 8.5: Value range and occupied memory using the data type »int« as an example.

The situation is different for the data type *byte*. Here the negative value range results from 2^7, the positive value range from 2^7 - 1. The fact that numerical data types have a sign is unfortunately not always practical. For many cases, only positive »natural« numbers with a value range from 0 to 255 would be necessary, which the data type *byte* does not provide.

The value range of numeric data types is based on the maximum value that can be realized on different computer systems. You remember: Java allows to write portable programs (chapter 2, »Technology Overview«). To achieve the portability of programs, the inventors of the language had to take into account what is feasible on different computer systems.

The highest representable amount of information is called »computer infinity« by experts. The computer infinity for many computer systems is 64 bits, which is why this also marks the limit of the Java number data types. The computer infinity in PC systems (with mathematical coprocessor), however, is 80 bits and is usually closed to Java programs, unfortunately.

By the maximum representable value range also a certain inaccuracy is unavoidable with some mathematical computations since no computer system is able to process all numbers arbitrarily exactly. With integer data types, the accuracy within the assured limits is always optimal. They are always stored completely as long as they are in the value range.

Floating point numbers, in contrast to integers, have in principle only a limited accuracy, even if they are in the value range of the data type. This is because the computer can store such numbers completely only if they have a limited number of decimal places.

When storing a floating point number, the computer breaks it into two parts. The first part is the exponent, and the second part is the mantissa; both are stored in binary. The decimal representation (Figure 8.6) shows that such a number is exact only up to a certain decimal place, everything else falls under the table. In this case, one speaks of so-called rounding errors.

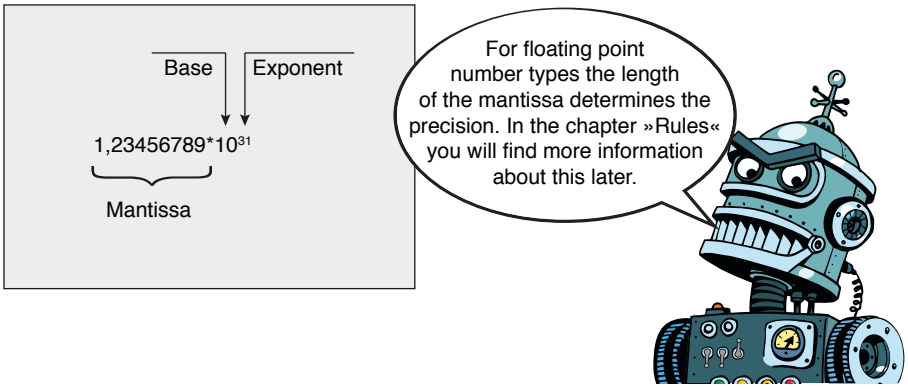


Figure 8.6: The digits of the mantissa determine the precision.

Exceed the Value Range

You must decide at declaration time whether the reserved value range and precision are sufficient for you to use a variable during the course of the program. If this is not the case and the variable exceeds its maximum valid value at some point or is too imprecise, program errors will occur. These can manifest themselves in different ways.

In the best case, program errors are noticed by a so-called overflow. In some cases, however, the program may go crazy and produce completely wrong values. In Chapter 23, »Rules«, you will learn exactly in which cases this happens and how you can protect your program from such conditions.

Data Type Selection

What does the risk of errors mean for the selection of a data type? First of all, it means that a software developer should consider very carefully at the programming stage which simple data type is suitable for a particular task based on its range of values.

- If the developer is too security-conscious and always uses too large data types, his program will run securely but consume too much memory.
- If the developer is too economical, his program will consume little memory, but it will not work properly.

In all cases of doubt, you should always prefer the first strategy.

Program Test

In any case, the developer must carefully consider which primitive data type is best suited for which part of the program, and he must test his program very carefully with respect to primitive data types (Chapter 23, »Rules«). How to use the data types is shown by some small application examples in the following sections.

8.3 Integer Data Types

Integers are mathematically correctly called fixed point numbers. The name says it all: They have no decimal places. They are used to represent numerical values from the natural number set. Java provides the four integer types *byte*, *short*, *int*, and *long*, which differ only in their value range (Figure 8.7).

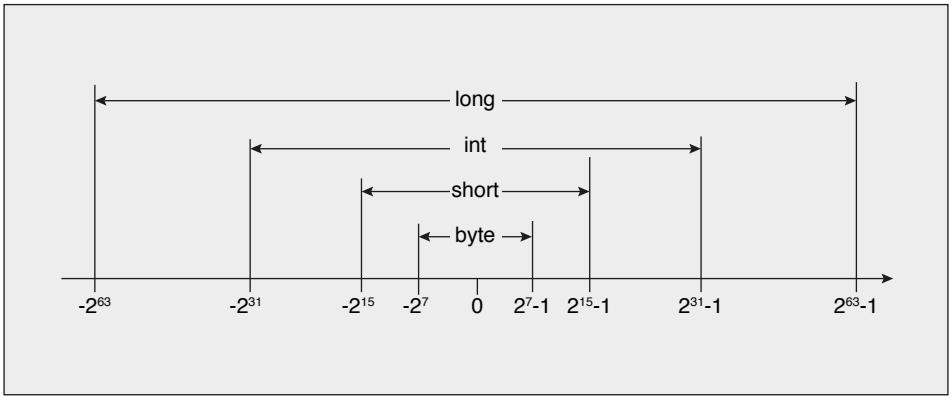


Figure 8.7: Value range of integer types.

8.3.1 Data Type »byte«

The data type *byte* has a sign like all integer types. It has a value range of only 8 bits, which corresponds to one byte, hence the name of the data type. With it you can only store numbers from $-2^7 = -128$ to $2^7-1 = +127$.

So for the following example program *Person* this data type is critical, because the height of a person 127 cm can easily be exceeded and thus would blow up the value range of *byte*. The example defines a class *Person* with an attribute *height* as data type *byte* (line 7). The class has a constructor that is passed by parameter with the height of the person (line 10). Furthermore, the class has a query method to convey the current value for *height*.

Listing 8.1: An example of using the data type »byte«.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    byte height; // Declaration
8
9    // Constructor:
10   public Person(byte height) {
11     this.height = height;
12   }
13
14   // Getter:
15   public byte getHeight() {
16     return height;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person tim = new Person((byte) 127);
22     System.out.println("Tiny Tim is " + tim.getHeight() + " cm tall");
23   }
24 }
```

The example program *Person* shown here produces the output:

```
Tiny Tim is 127 cm tall
```

8.3.2 Data Type »short«

The following applies to the data type *short*: it has the same length on all platforms, has a sign and a value range of 16 bits. Compared to the other data types, the value range is relatively short, hence its name.

Listing 8.2: An example of using the data type »short«.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    short height; // Declaration
8
9    // Constructor:
10   public Person(short height) {
11     this.height = height;
12   }
```

```
13
14    // Getter:
15    public short getHeight() {
16      return height;
17    }
18
19    // Start as a program
20    public static void main(String[] arguments) {
21      Person anna = new Person((short) 171);
22      System.out.println("Anna is " + anna.getHeight() + " cm tall");
23    }
24  }
```

This example also produces the output:

```
Anna is 171 cm tall
```

8.3.3 Data Type »int«

The *int* data type again doubles the value range of its predecessor to 32 bits. It is the standard data type for integer values in Java programs. Another example with the person *julia* shows how to use the data type (Listing 8.3).

Listing 8.3: An example of how to use the data type »int«.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    int height; // Declaration
8
9    // Constructor:
10   public Person(int height) {
11     this.height = height;
12   }
13
14   // Getter:
15   public int getHeight() {
16     return height;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person julia = new Person(172);
22     System.out.println("Julia is " + julia.getHeight() + " cm tall");
23   }
24 }
```

The program generates the output:

```
Julia is 172 cm tall
```

8.3.4 Data Type »long«

This data type increases the value range to the double of its predecessor and offers with 8 bytes (64 bits) the maximum value range for integers within a Java program (Listing 8.4).

Listing 8.4: An example of using the data type »long«.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    long height; // Declaration
8
9    // Constructor:
10   public Person(long height) {
11     this.height = height;
12   }
13
14   // Getter:
15   public long getHeight() {
16     return height;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person lukas = new Person(186);
22     System.out.println("Lukas is " + lukas.getHeight() + " cm tall");
23   }
24 }
```

This example also does not change the output of the program compared to the previous examples. Here the following output is produced:

```
Lukas is 186 cm tall
```

8.4 Floating Point Data Types

The counterpart of integers are the so-called floating point numbers. Unlike integers, these decimal numbers have decimal places. Mathematically speaking, they are used to process numerical values from the rational number set. Such numbers can be generated, for example, by a fraction calculation within a program. Java has two data types *float* and *double* for processing floating point numbers. The name *float* is derived from the English term for

floating point numbers. Because it doubles the value range of the data type *float*, it comes to the name *double* (Figure 8.8).

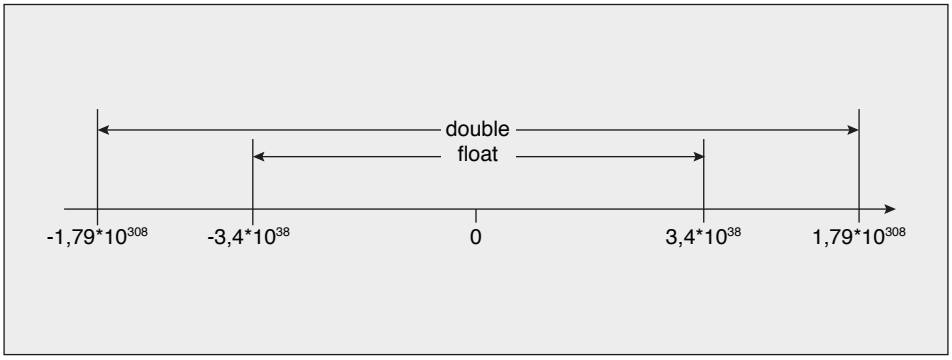


Figure 8.8: Value range of floating point data types (rounded values, non-linear representation).

8.4.1 Data Type »float«

The type *float* is the standard type for decimal numbers in Java programs and has a so-called single precision (32 bits). However, single precision is also only sufficient for simple arithmetic operations because the number of stored decimal places is small. An application is shown in Listing 8.5.

Listing 8.5: An example of using the data type »float«.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    float height; // Declaration
8
9    // Constructor:
10   public Person(float height) {
11     this.height = height;
12   }
13
14   // Getter:
15   public float getHeight() {
16     return height;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person florian = new Person((float) 1.85);
22     System.out.println("Florian is " + florian.getHeight() + " m tall");
```

```
  }
}
```

Because of the changed values for size, the following output is produced:

```
Florian is 1.85 m tall
```

8.4.2 Data Type »double«

The *double* type concludes the section on comma numbers of this chapter. You need this data type whenever the highest possible range of values and maximum precision for the decimal places must be expected. Listing 8.6 shows an example of its use again using the example of the height of a person.

Listing 8.6 An example of using the data type »double«.

```
//Examples/Primitive_Data_Types

package programmingcourse;

public class Person {

  double height; // Declaration

  // Constructor:
  public Person(double height) {
    this.height = height;
  }

  // Getter:
  public double getHeight() {
    return height;
  }

  // Start as a program:
  public static void main(String[] arguments) {
    Person robert = new Person(1.90);
    System.out.println("Robert is " + robert.getHeight() + " m tall");
  }
}
```

The program outputs the size of Robert as follows:

```
Robert is 1.90 m tall
```

8.5 Character Data Type

The character type *char* has been assigned a value range of 2 bytes, i.e. 16 bits. It uses the Unicode character set. *char* types are to be initialized with *simple* apostrophes (Listing 8.7). The character type is intended for outputting single characters. To output words, there is a more suitable data type in the *String* class (chapter 22, »Class Libraries«).

Listing 8.7: An example of char type usage.

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    char initial; // Declaration
8
9    // Constructor:
10   public Person(char initial) {
11     this.initial = initial;
12   }
13
14   // Getter:
15   public char getInitial() {
16     return initial;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person anna = new Person('A');
22     System.out.println("Anna's initial is an " + anna.getInitial());
23   }
24 }
```

The example program provides the following output:

```
Anna's initial is an A
```

8.6 Boolean Data Type

The predefined Java data type »boolean« can take the values *true* or *false*. As you will see later, such boolean values are used to control the flow of the program. In the following example, I would like to show only a very simple application of a truth value using the example of a person object. You may remember the example from chapter 3, »Object-Oriented Programming«. Here, the characters Anna and Julia could have a state that shows whether they are students.

I would like to show the example again with the robot Robert. A class *Person* should get the state *student* and be of type *boolean*. The result returned by a person object of this class can take two values: *true* or *false*. In the case of *true* the person is a student, and in the case of *false* it is not. The value is set to *false* at the beginning (line 21) via the constructor because Robert is not a student. To retrieve the value in line 22, the class has a query method called *isStudent()*. The example can be found below in Listing 8.8.

Listing 8.8: An example with truth values

```
1  //Examples/Primitive_Data_Types
2
3  package programmingcourse;
4
5  public class Person {
6
7    boolean student; // Declaration
8
9    // Constructor:
10   public Person(boolean student) {
11     this.student = student;
12   }
13
14   // Getter:
15   private boolean isStudent() {
16     return this.student;
17   }
18
19   // Start as a program:
20   public static void main(String[] arguments) {
21     Person robert = new Person(false);
22     System.out.println("Is Robert a student? " + robert.isStudent());
23   }
24
25 }
```

This example produces the following output:

```
Is Robert a student? false
```

8.7 Summary

Variables based on primitive Java data types are not objects. They are lightweight and require little memory. Because of their properties, they are primarily used as data types for the attributes of objects. In addition, they are suitable for a variety of tasks: The data type *char* stores single characters, while truth values can be used to specify simple states like *true* and *false*.

The data types *byte*, *short*, *int*, and *long* are intended for integers. Usually, for single precision integers, the *int* data type is taken, while numbers of the *long* data type can store

much larger numbers. If you want to use decimal numbers, you usually use the primitive data type *float*. Again, *double* is a floating point type that can store much larger values. Robert again summarizes the advantages of primitive data types (Figure 8.9).

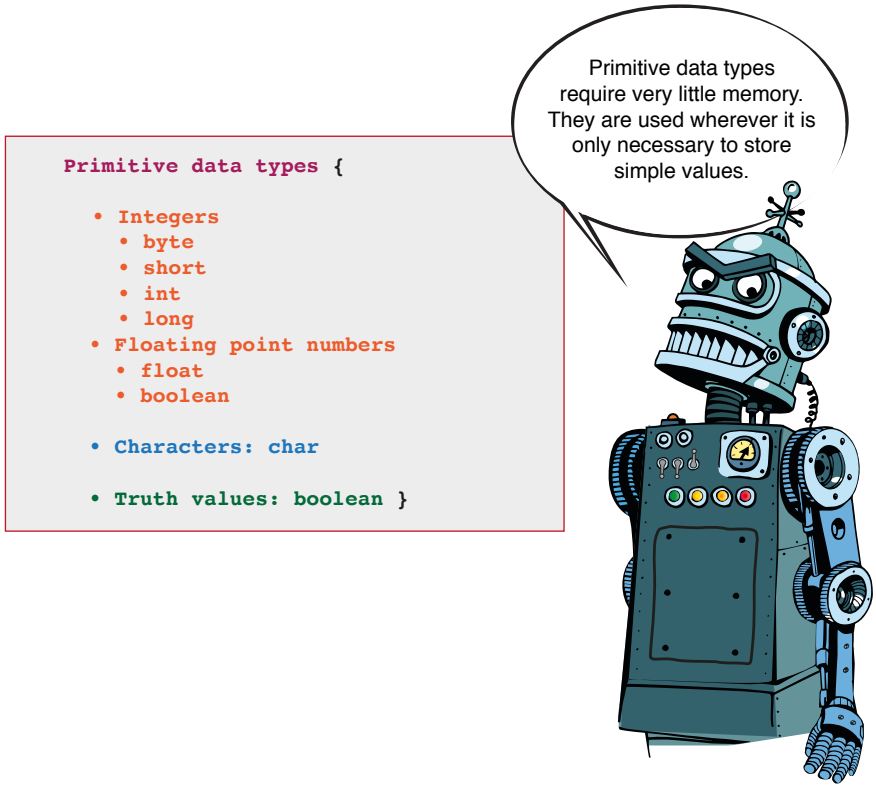


Figure 8.9: Primitive data types are lightweight.

This concludes this chapter, and you can reinforce your knowledge again in the following exercise.

8.8 Literature

Data types: *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html*

8.9 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/primitive_data_types

8.10 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge:
https://www.programmingcourse.net/courses/java_with_eclipse/primitive_data_types

There you will also find the solutions to the exercises.

9 Classes and Objects

9.1 Introduction

The program structure is introduced. You now know variables, statements and simple data types. Now you can really get started with the central topic of Java programming – with classes and objects. Do you still remember the definition of an object-oriented program by Alan Kay, the inventor of the programming language Smalltalk?

Figure 9.1: Classes are data types defined by the programmer of the class itself.

Alan Kay defined an object-oriented program as follows: »Everything is an object. Objects communicate by sending and receiving messages. Objects have their own memory. Every object is an instance of a class. The class holds the shared behavior for its instances. To eval a program list, control is passed to the first object and the remainder is treated as its message.«

9.2 Overview

9.2.1 Purpose of a Class

Java classes are templates for Java objects. A class brings together the commonality of many similar objects. You may remember the group of people from Chapter 3, »Object-Oriented Programming«. If you are looking for a common class for these person objects in Figure 9.2, you need to answer the question of what commonalities connect the different persons. So you need to *classify* the objects.

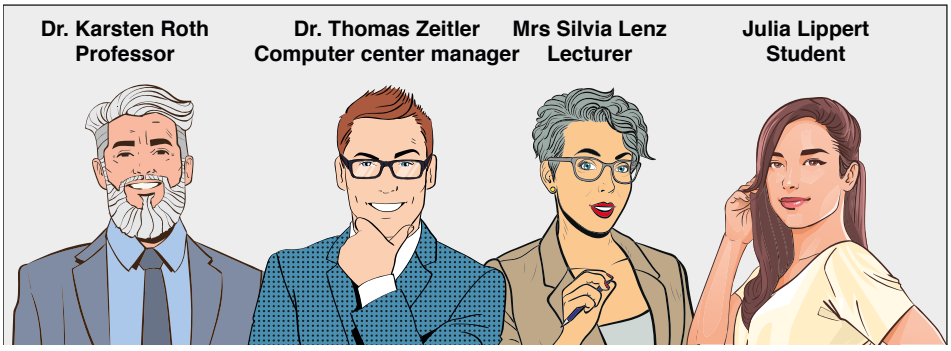


Figure 9.2: What do these person objects have in common?

The corresponding class should denote the commonalities of the persons in Figure 9.3. The solution in the form of a class always depends on the requirement one has for such a class. For example, a simple solution could look like the one shown in Figure 9.3.

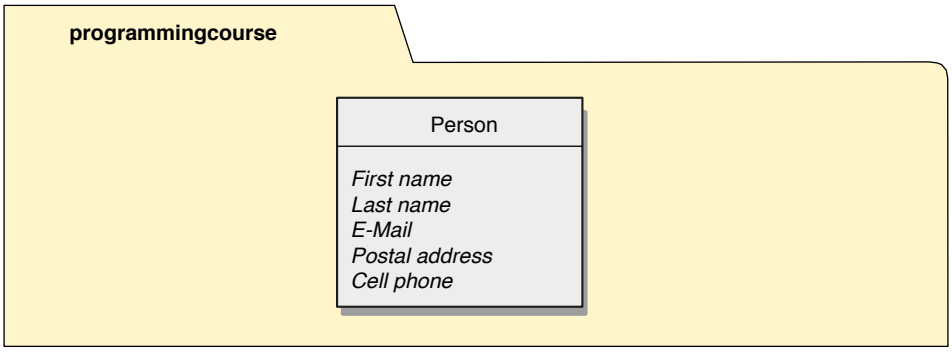


Figure 9.3: A class brings the commonality of many similar objects to a point.

In previous versions of Java, there was only one type of class. Because of the difficulty of adequately representing natural objects with this one type of class in Java programs, a number of additional types of classes have evolved over the course of the development of the Java programming language.

9.2.2 Types of Classes

In Chapter 3, »Object-Oriented Programming«, there was only ever talk of one type of class. Over time, the developers of Java have tried to meet all the design requirements of Java programs. Thus today there is not only one kind of class, but a whole set for the most different purposes: concrete and abstract classes, interfaces and generic classes (Generics). Of the concrete classes, there are also three other varieties: local, inner and anonymous classes (Table 9.1).

Table 9.1: Java has four main types of classes and three subtypes.

Class type	Expression	Usage
»Concrete« class	»Natural« class	General purpose class
	Local class	Auxiliary class
	Inner class	Auxiliary class with interface
	Anonymous class	Auxiliary class without name
Abstract class	-	Base class
Interface		Class without implementation
Generics (Generic class)	-	Type safe container class

Concrete classes are the natural form of a class, so to speak, as introduced in the chapter »Object-Oriented Programming«. Local classes are concrete classes that are defined within a class when it is not worthwhile to outsource them in a separate file. Similarly it behaves with internal classes, only that for this an interface is needed. Anonymous classes one defines likewise in a concrete class, but it carries no name.

Abstract classes and interfaces are intended for special design tasks in connection with concrete classes. Generic classes were introduced with Java 5 to avoid programming errors. As a Java beginner, this must seem quite confusing to you. The different classes do have their purpose, as the next pages will show.

You will use classes in Java programming that come from other programmers, but also classes that you have to define yourself. In both cases there is a so-called class definition. If you use a class like *String*, for example, this definition is included in the source code of the Java libraries. So you don't have to care about it. The situation is different for classes that you develop yourself. Here the first step is to define the class. Defining classes means that you specify the name of the class, its attributes, its methods and encapsulation.

9.2.3 Definition of Classes

Classes are among the freely definable data types. This means that the programmer of a class is free to define how his class should look like. For example, he can define the name of the class and determine which attributes and methods the class has. This is one of the large plus points of object-oriented programming. This allows classes to be built like real-world models, making the program much easier to understand. Figure 9.4 shows what a class definition looks like. The programmer of the class freely specifies in the class definition which components make up the class. This is done in a file corresponding to the name of

the class. The definition of a class consists of the name of the class preceded by the keyword *class*.

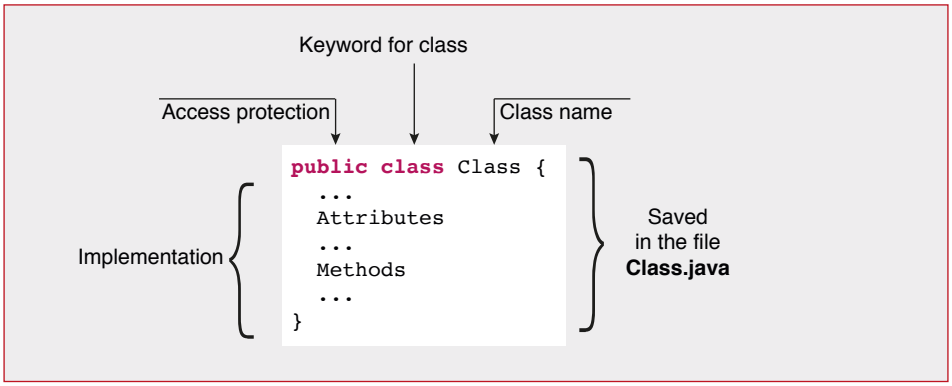


Figure 9.4: Classes must be defined before they can be used.

Additionally, the visibility of the class can be specified. The third part of this book introduces all keywords for the visibility of a class (chapter 23, »Rules«). At this point just this: the keyword *public* means that an object of this class is seen by every other object within a program. This is the normal case.

The implementation of the class contains attributes and methods. The technical term implementation means the implementation of the program. The implementation of the class begins with an opening curly bracket and ends with a closing curly bracket. The opening brace can also be on the next line after the class name.

9.2.4 Use of Classes

Once a class is defined, it »just« needs to be used. Using the class breaks down into two steps: Declaration and creation of an object.

Declaration

The declaration is identical to the variable declaration you already know from chapter 6, »Variables«. A variable is used to access an object. When declaring it, the program »tells« the compiler which data type this variable is of. In the case of objects, the data type is always the class from which the program creates the object. In the declaration, you first specify the type of the variable and then the name of the variable (Figure 9.5).

The declaration is necessary because Java is type-safe. This means that at compile time, each variable must have a data type. To avoid programming errors caused by assigning an incorrect type, the Java compiler takes strict care to avoid type violations throughout the program. Part of the stability of Java programs is based on the compiler's strict type checking.

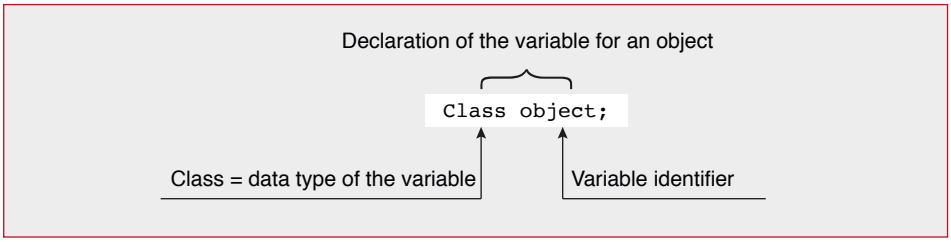


Figure 9.5: Objects are declared as variables.

Creation of Objects

Objects are created using the new operator. When creating, specify the new operator followed by the constructor of the class (Figure 9.6). Methods and operators will be discussed in detail in the next chapters. Only so much at this point: A method is nothing else than the designation of object-oriented programming for a function. Java operators are shorthand notations for functions, just as in mathematics. In this case, the function of the new operator is to create a new object using the constructor (often called instantiate).

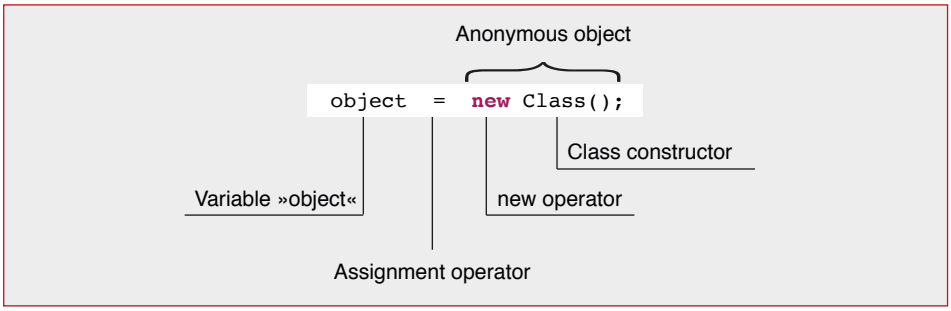


Figure 9.6: Objects are created with the new operator.

The constructor is called like a normal method when creating an object. What is special about the constructor is that it is the only method that is capitalized in Java. This allows it to be distinguished from other methods. It must also always be used together with the new operator As a result, you get an object whose name you store in a variable. It is very important to understand this process in detail. Already with the statement *new class()* an object is created. However, it has no label until you assign a variable to the object. Until that happens, the object remains anonymous, i.e. nameless. The variable gives you a reference to the nameless object and you can access it in the program.

Take a look at the table 9.2. The table represents the very simplified relationship between the variable *object* and the object from Figure 9.6 from a technical point of view. In the first column you can see the name of the data item, in the middle column its memory address and on the far right the contents of the memory. The representation is very simplified[1].

[1] To read the correct numeric memory addresses and complete memory contents of objects, you can use the open source tool JOL (Java Object Tool) (see literature at the end of the chapter)

Table 9.2 Very simplified relationship between variable and object.

Name	Memory address	Memory contents
-	Address of anonymous object	Anonymous object
Object	Address of variable	Reference to anonymous object

In the first line you can see what the *new Class();* statement did: it created an anonymous object of type *Class* at a given memory address. In very rare cases, this statement is left alone, because how could the object be accessed again without a variable? The assignment *object* = ... solves this problem. It assigns a variable to the result from *new Class();*. So the assignment stores the reference to the anonymous object in the variable named *object*.

Creation of Anonymous Objects

An object can also be created anonymously. That is, the object is not given a label - the assignment to a variable is omitted (Figure 9.7).

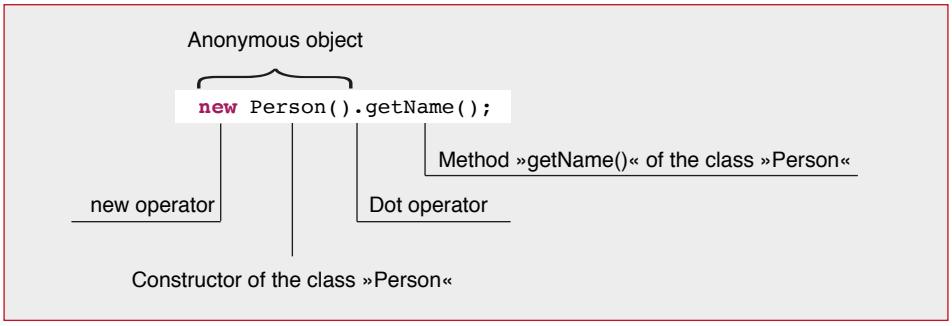


Figure 9.7: Creating an anonymous object and calling a method.

Of course, this means that you cannot call the object again in the further course of the program. The notation is therefore always used to call only a method.

Combination of Declaration and Definition

The separate notation from declaration and creation is always very practical when you do not want to create an object immediately. In many cases, however, the separate notation is not necessary and seems rather cumbersome to many programmers. Therefore, declaration and creation can also be combined and written in one statement (Figure 9.8).

9.3 Anonymous Classes

9.3.1 Definition of Concrete Classes

As with all Java class types, the concrete class is one of the data types that can be freely designed.

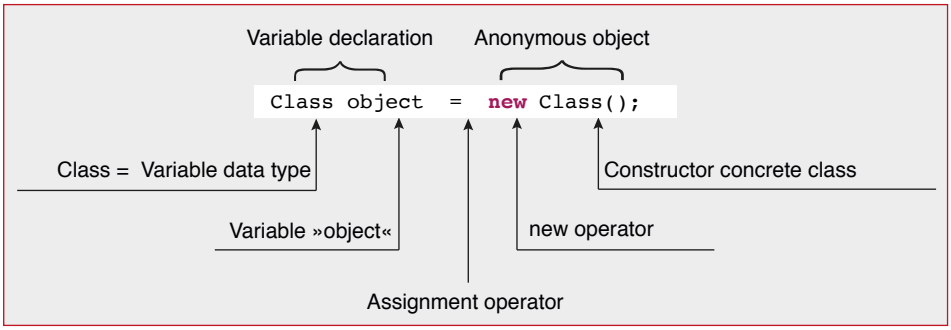


Figure 9.8: Combination of declaration and creation of an object.

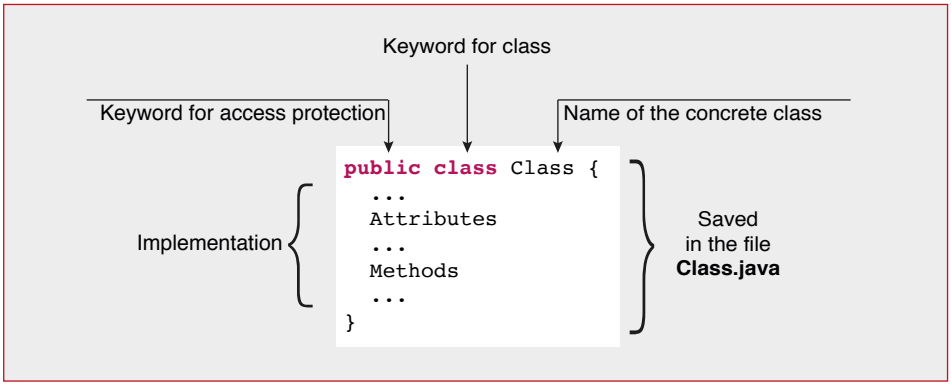


Figure 9.9: Classes must be defined before you can use them.

As a Java example of a concrete class, I'll go back to the class *Person* that you already know from the chapter »Object-Oriented Programming«. It had the attributes *name, height, hairColor* and *student*. In Java, this class is defined as shown in Listing 9.1.

Listing 9.1 The definition of the class »Person« with its attributes.

```
1 public class Person {
2   private String name;
3   private int height;
4   private String hairColor;
5   private boolean student;
6 }
```

So, since a class is stored in a file with the same name, the file name in this example is *Person.java*. What you will notice in listing 9.1 is that the attributes are written in lower case. The background to this deviation from the chapter 3, »Object-Oriented Programming«, is Java naming conventions. Classes must be capitalized in Java, all keywords, variables and methods – except the constructor – must be lowercase.

The background of this mixed upper and lower case is the better readability of the source code. As long as you work with a development environment like Eclipse, you don't have

to learn these rules by heart, of course. The Eclipse development environment always reminds you to follow the naming conventions when you create a new class, object, or method.

9.3.2 Creation of Objects of a Concrete Class

Declaration

There are two stages to creating an object of a concrete class: First, you must declare an object (Figure 9.10), and only then do you create it.

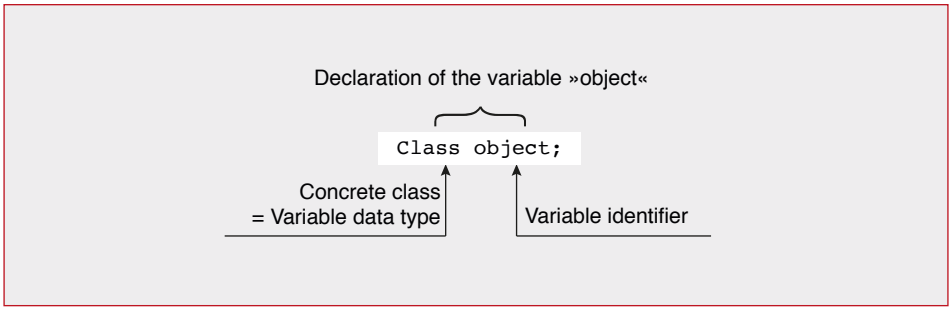


Figure 9.10: Declaration of an Object.

The declaration using the class *Person* as an example will look like 9.11. Note that the declaration is terminated with a semicolon and the variable *anna* is lowercase.

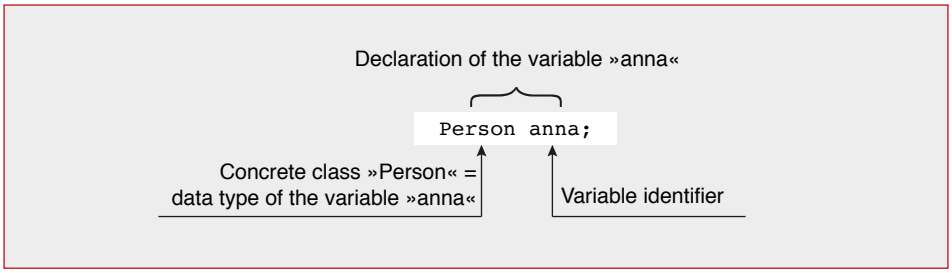


Figure 9.11: Example of the declaration of object »Anna«.

Creation

You create a new object from a concrete class using the *new* keyword and calling the constructor of the class from which the object is to be created (Figure 9.12).

In a Java program, the creation of the object from the example *anna* looks like figure 9.13.

As said at the beginning, the declaration and creation of an object from a concrete class can also be combined. In a Java program, using the object *anna* as an example, this looks like in Figure 9.14. On the left side of the assignment you can see the declaration of the variable *anna*. In it is stored the reference to the object that will be created on the right side.

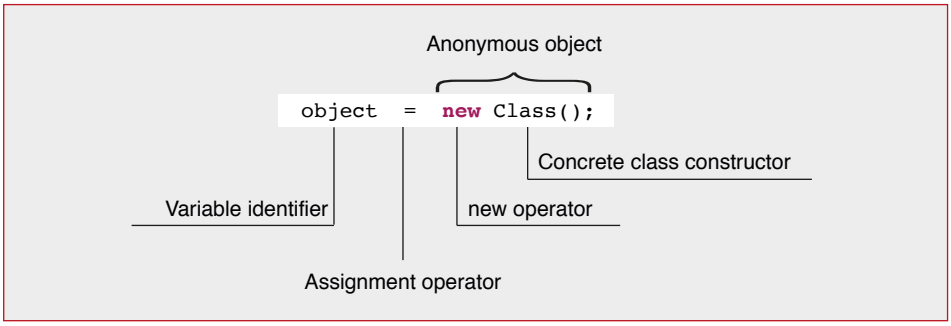


Figure 9.12: Creation of an object from a concrete class.

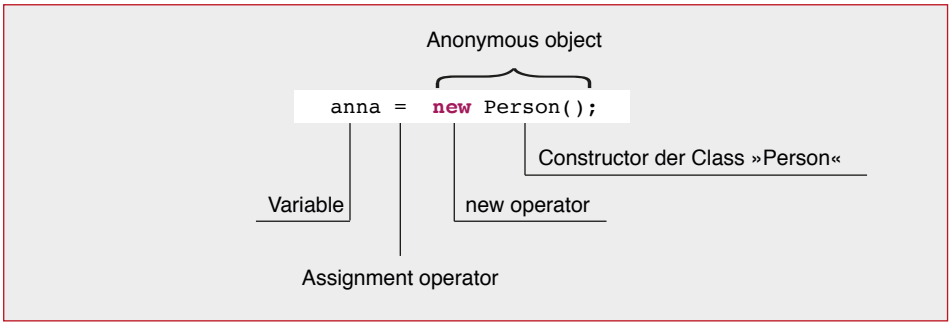


Figure 9.13: Creation of the object anna from a concrete class.

9.3.3 Inner Classes

Sometimes it is not worth having a separate file for a class – especially for auxiliary classes that cannot be reused. In this case there are so-called inner classes. They are called so because you define them inside a concrete class (Figure 9.15). Here you can freely choose the visibility of the class to the outside (*public, default, protected, private*).

In Listing 9.2, you can see the inner class *foundation* as an example. Only the robot itself can access this class and execute the method *sayGoodMorning(String name)*. The inner class consists of only one method, for which an independent class in a separate file would be excessive.

Listing 9.2: An example of an »inner class«.

```
1  public class Robot {
2
3    // Constructor:
4    public Robot() {
5      // Creation on basis of the inner class
6      Greeting greeting = new Greeting();
7      greeting.sayGoodMorning("Anna");
8    }
9
```

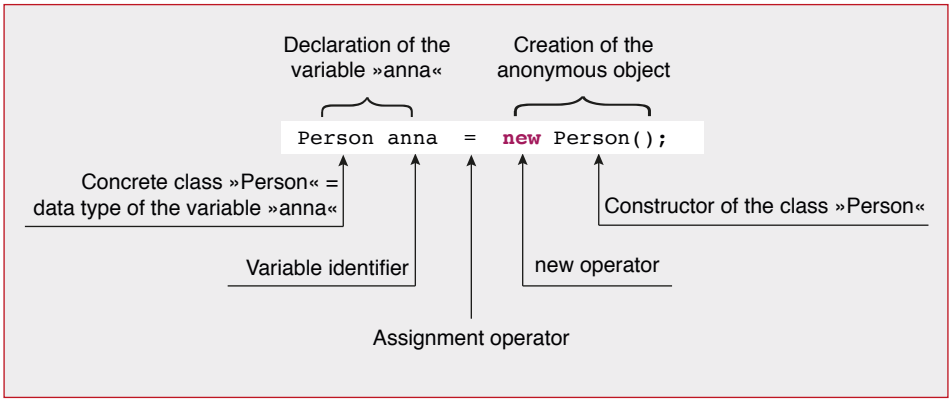


Figure 9.14: Combination of declaration and creation of the concrete object *anna*.

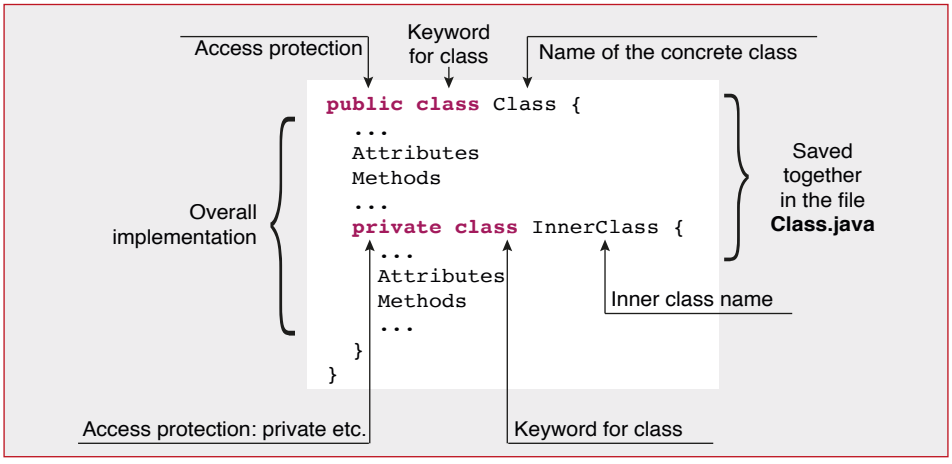


Figure 9.15: Definition of an inner class.

```
10    // Definition of the inner class "Greeting"
11    private class Greeting {
12      public void sayGoodMorning(String name) {
13        System.out.println("Good morning, " + name + "!");
14      }
15    }
16  }
```

The *InnerClassDemo* program creates a new robot object (listing 9.3).

Listing 9.3: The program »InnerClassDemo« creates a new robot object.

```
1  public class InnerClassDemo {
2
3    public static void main(String[] arguments) {
4      Robot robert = new Robot();
```

```
5     }
6  }
```

Using the program, Robert prints the greeting you see in Figure 9.16.

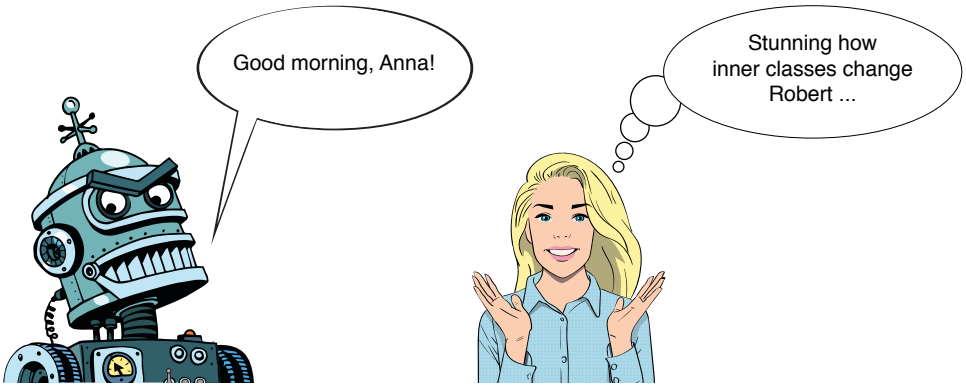


Figure 9.16: Robert greets Anna with the help of an inner class.

9.3.4 Local Classes

Local classes are a subtype of the inner class. They are defined within a block of a concrete class (Figure 9.17). Unlike the inner class, only the keywords *abstract* or *final* are allowed for this local class. The keyword *abstract* represents an abstract class (Section 9.4, »Abstract Classes«). The keyword *final*, on the other hand, means that this class does not inherit anything.

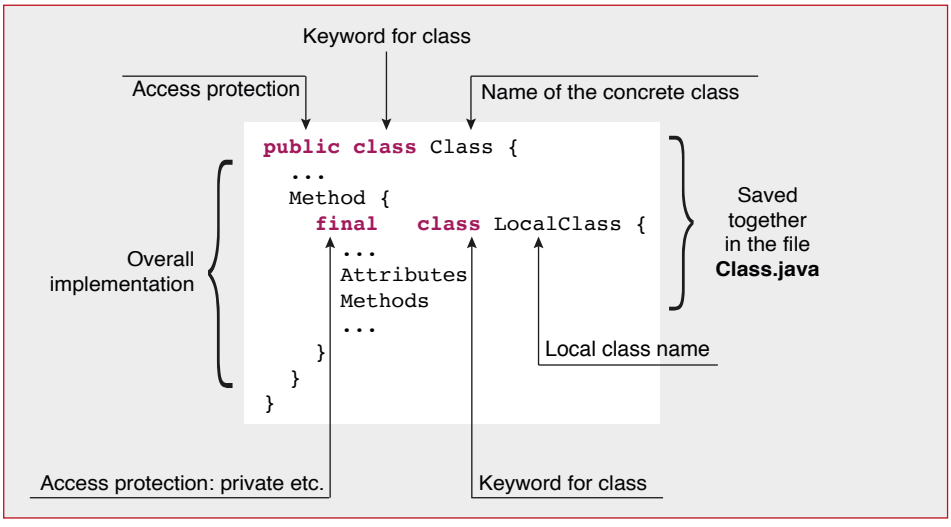


Figure 9.17: Definition of a local class.

In Listing 9.4 you can see the local class *Greeting* as an example. Only the robot itself can access this class and execute the method *sayGoodMorning()*. The local class consists of only one method, for which an independent class in a separate file would be excessive.

Listing 9.4: This time the robot's greeting is an example of a »local class«.

```
1  public class Robot {
2
3    public void sayGoodMorning(String name) {
4      // Definition of the local class "Greeting"
5      final class Greeting {
6        Greeting(String name) {
7          System.out.println("Good␣morning,␣" + name + "!");
8        }
9      }
10     // Creation with the local class
11     new Greeting(name);
12   }
13 }
```

9.3.5 Anonymous Classes

Another form of auxiliary classes that are defined inside a concrete class is called an anonymous class. Unlike the local classes just mentioned, this species has no name. The anonymous class is created from an interface. The section 9.5, »Interfaces« discusses interfaces in detail. At this point, just this much about interfaces: They are abstract classes without implementation. They are used to specify the implementation. When implemented, the anonymous class overrides the default methods of the interface (Figure 9.18).

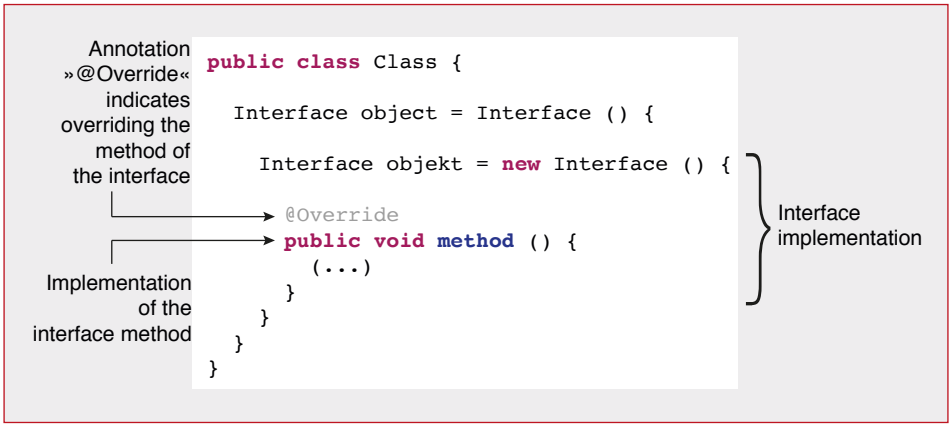


Figure 9.18: Definition of an anonymous class.

The Listing 9.5 shows a class created from the class *Greeting* that does not have a name. As you can see in line 5, the statement *new Greeting()* does not end with a semicolon, as

usual, but opens a block via a curly bracket. In line 6 follows a so-called annotation, which informs the compiler that the following implementation overwrites the method of the underlying interface *greeting*. The closing bracket on line 10 ends the class definition.

Listing 9.5: Example of an anonymous class.

```
1  public class Robot {
2
3    public void sayGoodMorning(String name) {
4
5      Greeting greeting = new Greeting() {
6        @Override
7        public void sayGoodMorning(String name) {
8          System.out.println("Good morning, " + name + "!");
9        }
10     };
11     greeting.sayGoodMorning(name);
12   }
13 }
```

The anonymous class implements the interface *Greeting* in line 5. The interface is simple in structure and consists only of a wrapper for the method *sageGoodMorning(String name)* (Listing 9.6).

Listing 9.6: The anonymous class implements this interface.

```
1  public interface Greeting {
2
3    void sayGoodMorning(String name);
4
5  }
```

In the case of the class *Greeting*, the interface specifies to implement a method named *sayGoodMorning(String name)*. The interaction between all the elements introduced so far is shown by the program *AnonymousClassDemo* (9.7).

Listing 9.7: The anonymous class implements this interface.

```
1  public class AnonymousClassDemo {
2
3    public static void main(String... args) {
4      Robot robert = new Robot();
5      robert.sayGoodMorning("Julia");
6    }
7  }
```

The program creates an object of type *robot* with the name *robert* in line 4. Robert again greets Anna by calling the *sayGoodMorning("Julia")* method in line 5, after which the program outputs the greeting (Figure 9.19).

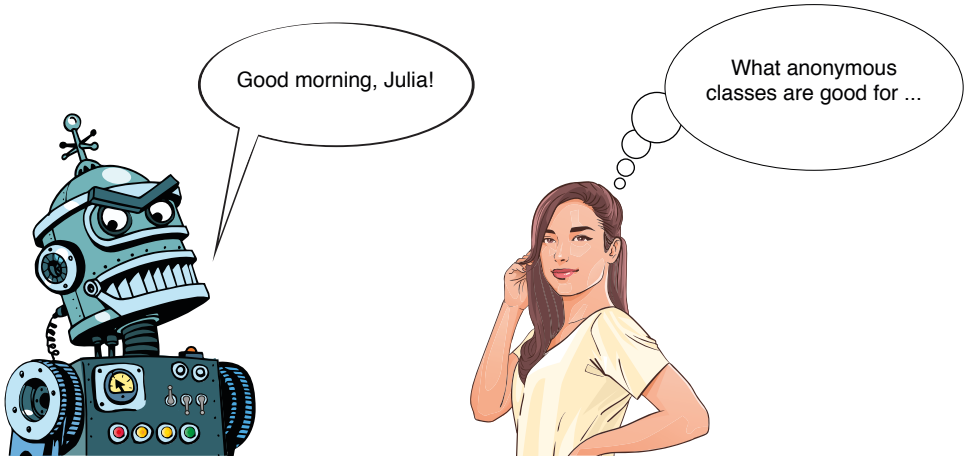


Figure 9.19: Robert greets Julia with the help of an anonymous class.

9.3.6 Inheritance

Keyword »extends«

When you want to inherit a class, you extend it with certain properties. The keyword for inheritance is *extends* for concrete classes accordingly. As an example I would like to take up again the class essence, which you already know from chapter 3. A class *Being* is to be defined, which serves as a base class for humans and robots. The following diagram shows the inheritance relationship between base class and derived classes (Figure 9.20).

In Listing 9.8 you can see the implementation of the base class *Being*. It contains only the *name* attribute, a constructor that creates the object, and the method *getName* that returns the name of the entity.

Listing 9.8: The base class »Being« lays the foundation for the two derived classes.

```
1  public class Being {
2
3    private String name;
4
5    // Constructor with the attribute name
6    public Being(String name) {
7      this.name = name;
8    }
9
10   /**
11    * Returns the name of the object
12    * @return name
13    */
14   public String getName() {
15     return name;
16   }
```

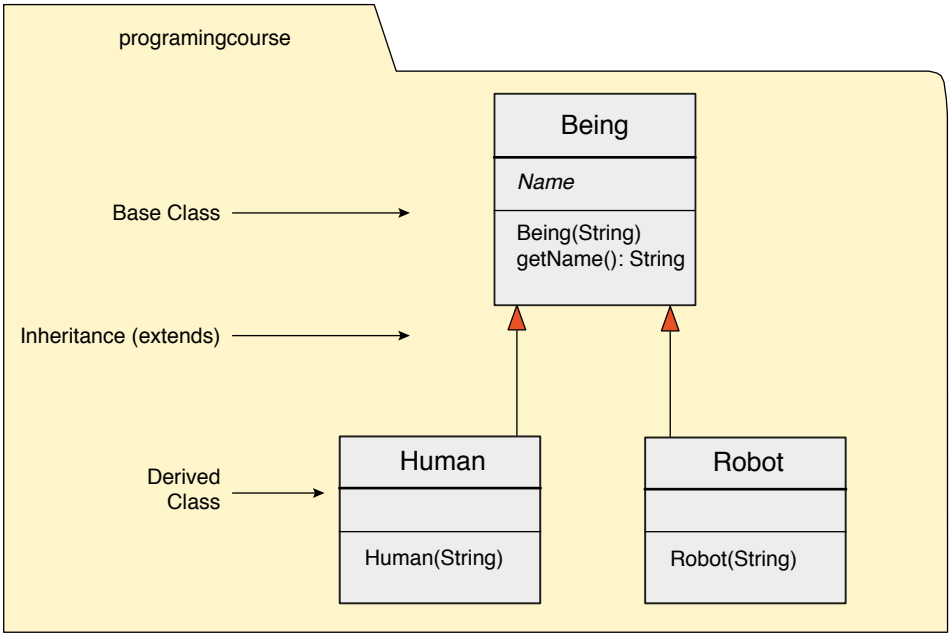


Figure 9.20: The classes »Robot« and »Human« extends the base class »Being«.

.

The base class *Being* is extended by the class *Human*. Or, with the human point of view: The class *Human* inherits from the base class *Being*. It is noticeable that the class contains nothing except the constructor.

Listing 9.9: The class »Human« extends the class »Being«.

```
1  //Examples/Classes_and_Objects
2
3  package programmingcourse;
4
5  class Human extends Being {
6
7    // Constructor:
8    public Human(String name) {
9      super(name);
10   }
11 }
```

The base class *Being* is also extended by the class *Robot*. Again, using the robot's point of view, we can say that the class *Robot* inherits from the base class *Being*. This class also does not contain any implementation except for the constructor.

Look at the application of the two classes using the program *InheritanceDemo*. First, the program creates a new human, an object called *anna*, in line 8, then a robot, an object called *robert*, in line 9. In line 10 the program calls the method *getName* from the object *anna*. As you have seen in the source code of the class »Human«, this method does not

exist there. It is inherited through the class *Being*. Analogously, the statement in line 11 behaves. There, too, the program uses the *getName* method, which was defined only in the base class *Being*.

Listing 9.10: This program demonstrates the use of inheritance.

```
1  public class InheritanceDemo {
2
3    public static void main(String[] arguments) {
4      Human anna = new Human("Anna");
5      Robot robert = new Robot("Robert");
6      System.out.println("The human is called " + anna.getName());
7      System.out.println("The robot is called " + robert.getName());
8    }
9  }
```

At this point, you may have noticed the keywords *this* and *super*, which play a big role in Java programming.

Keyword »this«

You already know the keyword *this* as an object variable of a class, which the compiler creates automatically. It always »points« to the object from which it is called. For this again the example with the class *Being* (Figure 9.21).

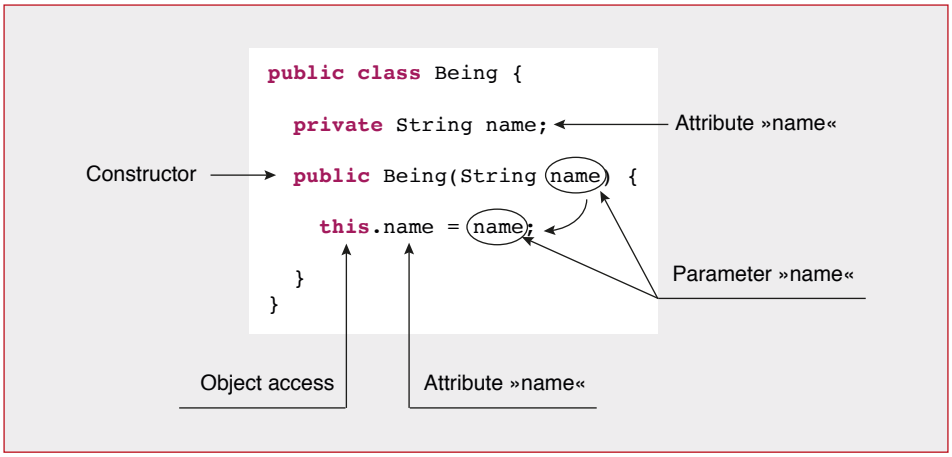


Figure 9.21: Use of keyword »this«.

From the figure, you can see that the name of an entity is passed via the *name* parameter of the constructor. For example, the name could be *robert*. The parameter has the same name as the attribute in this program, which doesn't bother at all. To distinguish between the two same names for the parameter and the attribute, the program uses the pointer *this*. With the statement

```
this.name
```

the program addresses its own attribute *name*. With the statement

```
this.name = name;
```

the program passes the **attribute** *name* on the left the value of the **parameter** *name* on the right.

Keyword »super«

Similar to the *this* variable, the *super()* method is an integral part of every class. If you do not write method explicitly in the constructor of a class, the compiler generates this call. You must include the method as the first statement in the constructor. With its help you access the constructor of the base class. The base class is called a superclass in English - hence the name of the method. If you use *super()* without addition, you call the constructor without parameters of the base class. However, it is important to note that you must explicitly call *super()* with parameters. Using the example of the constructor with one parameter is the most common usage and can be seen in Figure 9.22.

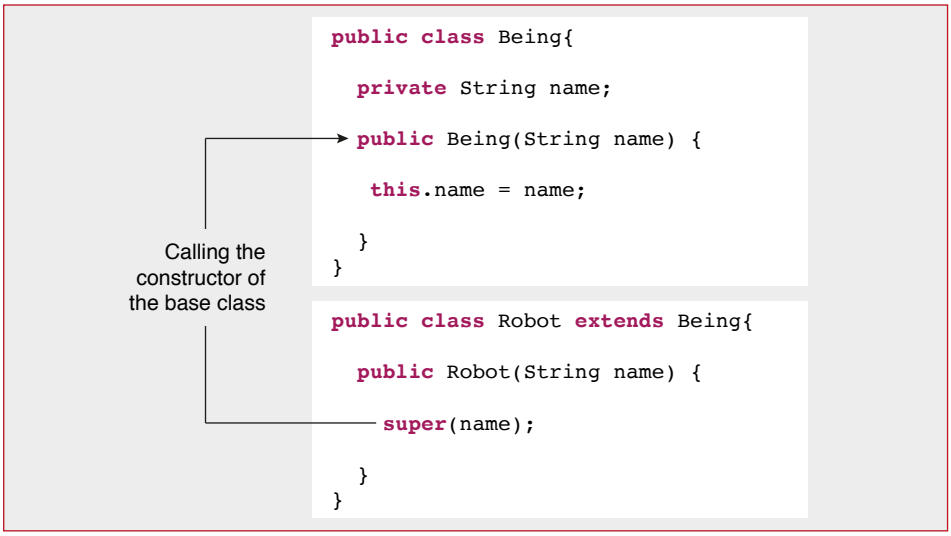


Figure 9.22: With »super« you access the constructor of the base class.

You recognize here again the base class *Being* with its constructor. The derived class does not have its own attribute *name*. It passes this to the base class via the call using the keyword *super()*. In a Java program, this looks concretely like this for the base class *being*, as seen in Listing 9.11.

Listing 9.11: The class »being« is the base class.

```
5  package programmingcourse;
6
7  public class Being {
8
9    String name = "Being";
```

```
10
11    public Being(String name) {
```

The derived class *robot* has no attributes of its own. To set the *name* attribute, it infuses this class via a parameter of the same name in the base class in line 4 (Listing 9.12).

Listing 9.12: The class »Robot« extends the class »Being«.

```
1  public class Robot extends Being {
2
3    public Robot(String name) {
4      super(name); // Call of the constructor of the base class
5    }
6  }
```

The interaction of the two classes comes into play in the *SuperDemo* program (listing 9.13).

Listing 9.13: This program shows the use of the keyword »super«.

```
1  public class SuperDemo {
2
3    public static void main(String[] args) {
4      Robot robert = new Robot("Robert");
5      System.out.println("I am a robot called " + robert.getName());
6    }
7  }
```

The program creates a new robot object named *robert* and passes it the new name »Robert« using the *name* parameter. On line 5, this new object prints its name. When you run the program, you get the following output:

```
I am a robot called Robert
```

You can learn more about the »super method« in the exercises at the end of the chapter.

Keyword »final«

To prevent a class from being inherited, you put the keyword *final*, which you already know from constants, before the name of the class. Here it has a slightly different effect: no subclasses can be created from classes that are marked in this way. This is true of many classes that are part of the Java standard library, such as the class *String*, a snippet of which you can see in Figure 9.23.

The point behind the »inheritance prohibition« is that it can be used to prevent class hierarchies. Class hierarchies with very many levels lead to very strong dependencies in programs. If one changes a detail at a basis class, one must change often a long chain of classes. This leads to very high test expenditure and is accordingly expensive. With the keyword *final* such long inheritance hierarchies can be prevented.

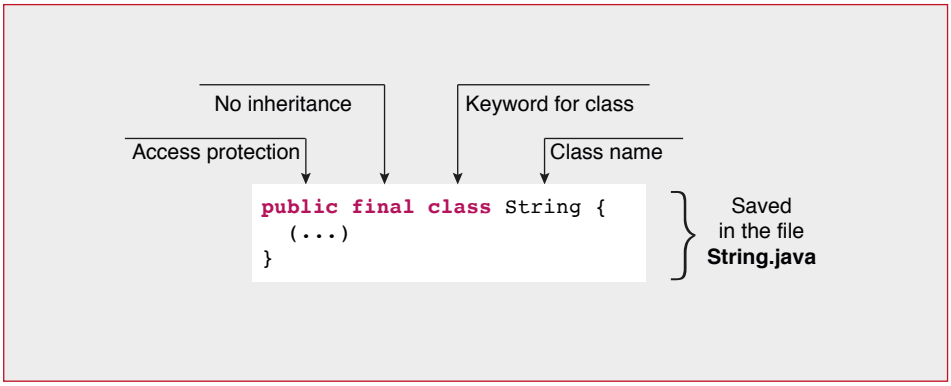


Figure 9.23: Definition of a final class.

9.4 Abstract Classes

Now you know how to inherit classes and how to prevent classes from being inherited. You also know what inner and local classes are and how to call components of parent classes with *super*. What is still missing? What's missing is preventing objects from being created from a class. This can be achieved with abstract classes (Figure 9.24).

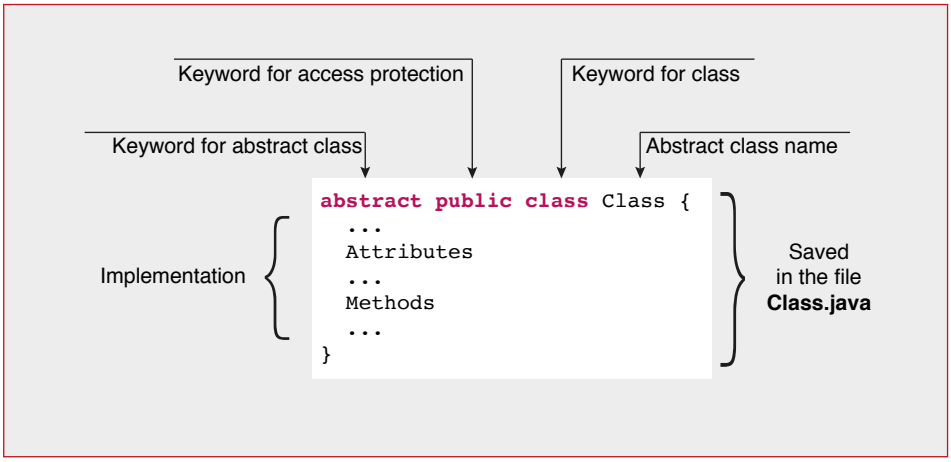


Figure 9.24: You cannot create objects from an abstract class.

Here's an example: The class *being* of the example defined in the previous example signals by its very name that there are no concrete objects. But to really avoid creating objects, you have to define the class as *abstract* (Listing 9.14).

Listing 9.14: The class »Being« as an abstract class.

```
1 abstract public class Being {
2   private String name;
```

```
3
4     public Being(String name) {
5       this.name = name;
6     }
7
8     public String getName() {
9       return name;
10    }
11  }
```

Of course, classes can be derived from an abstract class, because the keyword *final* is not allowed here, otherwise you could not do anything with the class. Derived classes use *extends* to extend the class (Listing 9.15).

Listing 9.15: The class »Robot« extends the abstract class.

```
1  class Robot extends Being {
2
3    public Robot(String name) {
4      super(name);
5    }
6  }
```

The program *AbstractClassDemo* again shows the interaction of the individual components abstract class, concrete class and the creation of an object within a program (Listing 9.16).

Listing 9.16: This program demonstrates the use of the abstract class »Being«.

```
1  public class AbstractClassDemo {
2
3    public static void main(String[] arguments) {
4      Robot robert = new Robot("Robert");
5      System.out.println("Hello, I am " + robert.getName() + "!");
6    }
7  }
```

The program outputs the message you see in Figure 9.25.

9.5 Interfaces

An »interface« is a special form of an abstract class. It is a collection of abstract methods and constants. The interface does not contain any constructors, and therefore there are no objects of it. An interface always requires a derived or anonymous class that implements *all* must implement (fill with life) all methods of the interface. An interface can be used to implement multiple inheritance, which is not available in Java itself. How an interface is defined is shown in Figure 9.26. Unlike all other classes, the keyword is simply *interface* to define an interface.

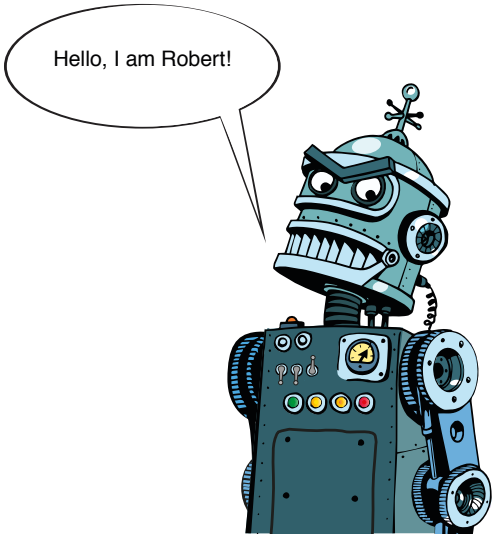


Figure 9.25: Robert extends the abstract class »Being«.

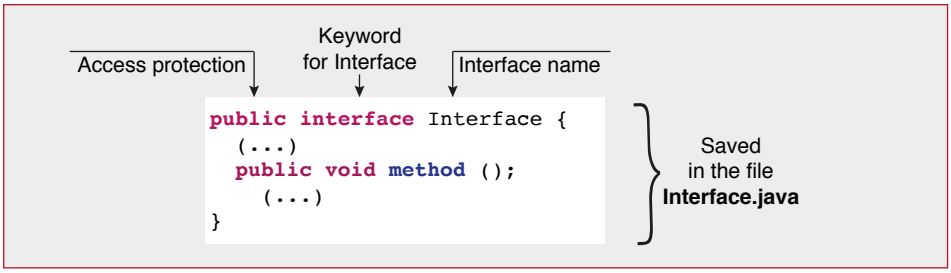


Figure 9.26: Definition of an interface.

There are three important reasons to use interfaces: Encapsulation of components, realization of multiple inheritance, and grouping of identical methods. Components form an encapsulation around multiple classes whose interfaces should not be fully exposed to the outside world. An interface offers here a subset of the internal interfaces. Multiple inheritance was not realized in Java because of the disadvantages mentioned. Nevertheless, it can be important for design reasons to inherit properties from several classes. This is exactly the purpose of interfaces. An example of how an interface can be implemented is shown in Figure 9.27.

Here the interface is implemented by a normal class. Accordingly, the keyword is not *extends*, but *implements*. The derived class must implement the interface of the interface in the form of the method *method()*.

Another form of implementation is possible via an anonymous class. In this case, the interface is called with the new operator, which is followed by an anonymous class. The definition of the anonymous class contains the implementation of the method(s) of the interface (Figure 9.28).

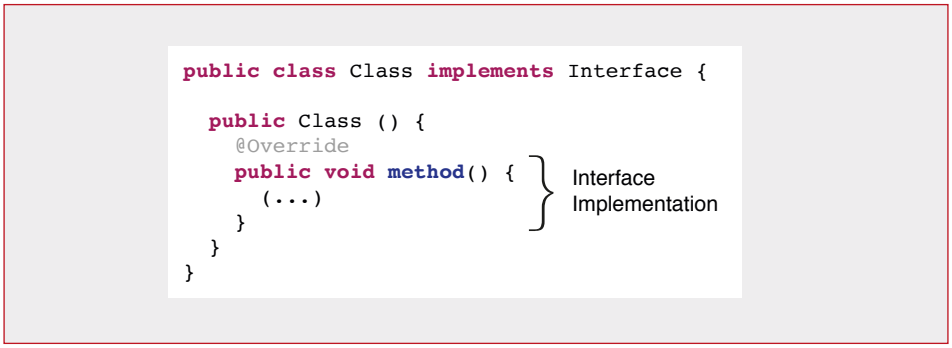


Figure 9.27: Implementation of an interface using a »natural« class.

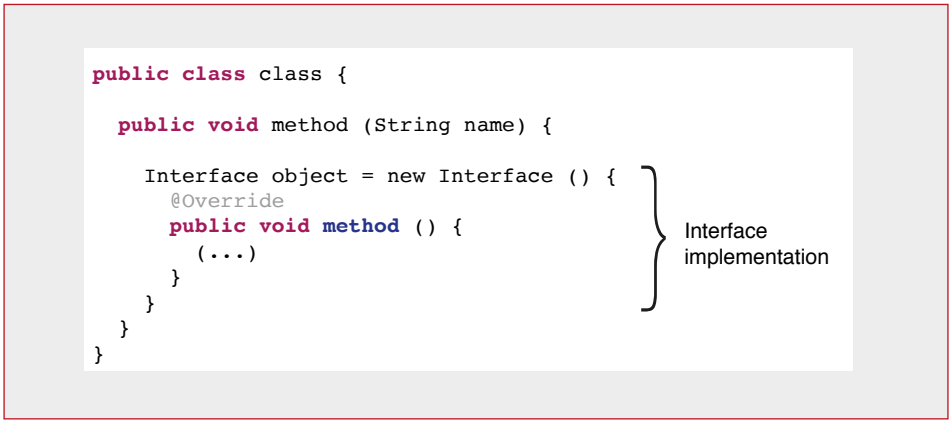


Figure 9.28: Implementation of an interface using an anonymous class.

To show the use of interfaces a bit more vividly, I would like to introduce the implementation using an anonymous class with an example. For this purpose, the class *greeting*, which you already know from the previous examples, is transformed into an interface. The interface *Begruessung* has only the method *sageGutenMorgen* (Listing 9.17). All classes that use the interface must implement the method.

Listing 9.17: The class »Greeting« as interface

```
1 public interface Greeting {
2
3   public void sayGoodMorning(String name);
4
5 }
```

As a small variation on the last »Greeting«, the class *Robot* uses the interface multiple times. It implements three different greetings in three different languages. In lines 5, 12, and 19, the class creates three anonymous classes each. Line 5 shows the implementation of a German greeting, line 12 that of an English one, and line 19 that of the Spanish one (Listing

9.18). In all three cases, the anonymous class must override the *sageGutenMorgen(String name)* method.

Listing 9.18: The class »Robot« implements the interface »Greeting«.

```
1  public class Robot {
2
3    public void sayGoodMorning(String name) {
4
5      Greeting englishGreeting = new Greeting() {
6        @Override
7        public void sayGoodMorning(String name) {
8          System.out.println("Good morning, " + name + "!");
9        }
10     };
11
12     Greeting germanGreeting = new Greeting() {
13       @Override
14       public void sayGoodMorning(String name) {
15         System.out.println("Guten Morgen, " + name + "!");
16       }
17     };
18
19     Greeting spanishGreeting = new Greeting() {
20       @Override
21       public void sayGoodMorning(String name) {
22         System.out.println("Buenos dias, " + name + "!");
23       }
24     };
25
26     englishGreeting.sayGoodMorning(name);
27     germanGreeting.sayGoodMorning(name);
28     spanishGreeting.sayGoodMorning(name);
29   }
30 }
```

Finally, the program*InterfaceDemo* brings all the pieces back into play with each other. First, the program creates an object of type *robot* in line 4. In the next line, this robot greets the student Julia in three languages thanks to the triple implementation of the interface in the class *Robot* (Listing 9.19).

Listing 9.19: This program shows the use of an interface.

```
1  public class InterfaceDemo {
2
3    public static void main(String... args) {
4      Robot robert = new Robot();
5      robert.sayGoodMorning("Julia");
6    }
7
8  }
```

The interaction results in Julia being greeted by the robot Robert in a slightly different way than usual this time. Julia does not seem very pleased about Robert's transformation (Figure 9.29).

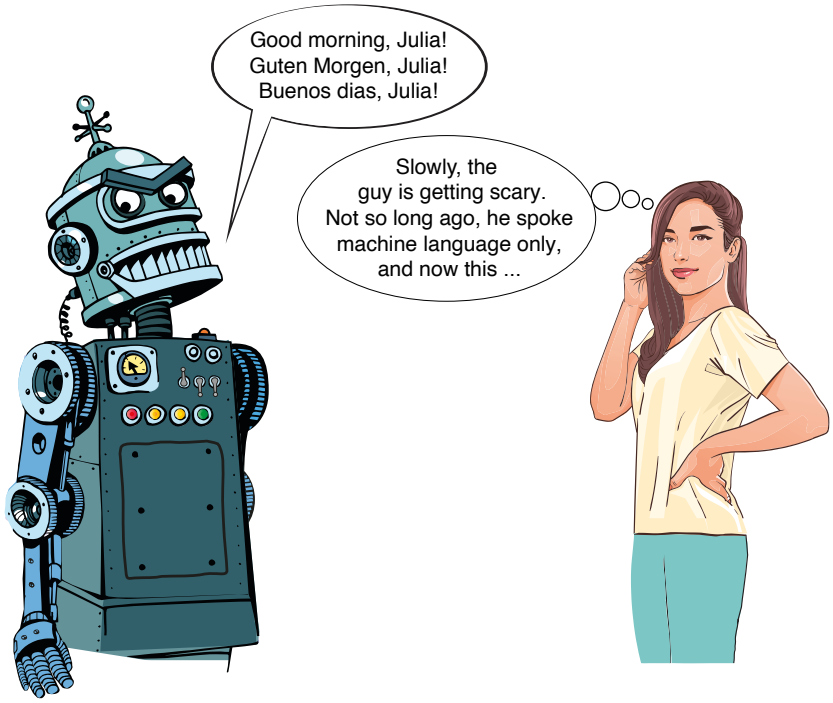


Figure 9.29: Robert now greets Julia in three languages.

9.6 Generics

The developers of the Java programming language introduced generics with Java 5. The reason for generic classes was that prior to Java 5, the language was not designed to be completely type safe. In certain cases, the programmer could assign variables of the wrong type when making an assignment. The Java compiler was not able to detect such errors. You will learn how this could happen in a moment. But first, take a look at what the definition of a generic class looks like.

9.6.1 Definition of Generic Classes

As with the other class types, you can design a generic class yourself. However, this requires that you first define the class before you can use it . This is done in a file that corresponds to the name of the class. Figure 9.30 shows how the definition of a generic class looks like

using a concrete class as an example. However, you can define not only concrete classes, but also generic interfaces, for example.

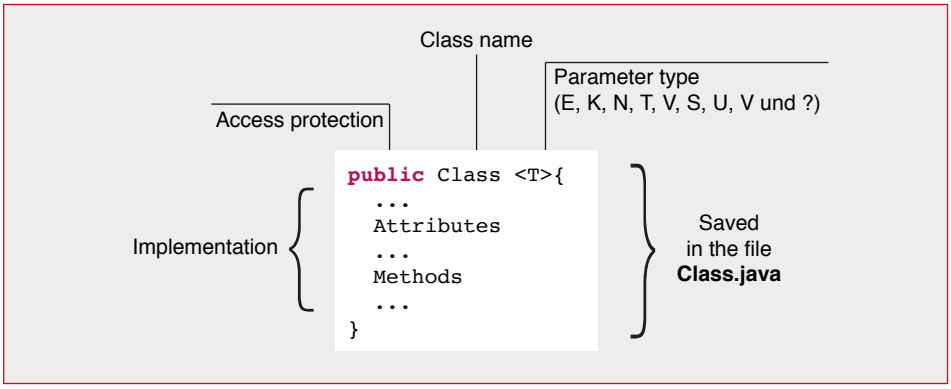


Figure 9.30: The definition of a generic class.

Table 9.3 shows which types of parameters can be passed to a generic.

Table 9.3: Types of parameters.

Parameter	Label	Meaning
E	Element	Element
K	Key	Key
N	Number	Number
T	Type	Data type
V	Value	Value
S,U,V	Number	Numeric value

9.6.2 Creation of Generic Objects

Objects of type *Generics* are created quite similar to what you know from concrete classes. However, the syntax is a bit more complex (Figure 9.31).

Suppose you want to develop a robot that is always imprinted on a particular person and receives commands only from that person. However, you do not know what type this person is. For example, it could be a lawyer, engineer, musician, student, or professor. Not an everyday task. How would you proceed?

It would be clumsy to develop a separate robot for each type of person. Instead, you need a universal model. Suppose the people who can order a robot are professors and students. Then first define a simple class *Professor* with a name as an attribute, a constructor, and an accessor method (Listing 9.20).

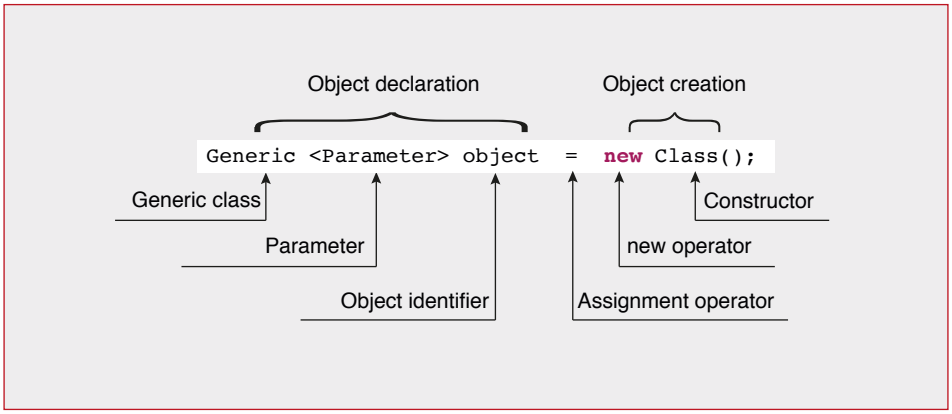


Figure 9.31: The creation of an object from a generic class.

Listing 9.20: The class »Professor« consists of attribute, constructor and access method.

```
1  public class Professor {
2    //Attribute:
3    private String name;
4    //Constructor:
5    public Professor(String name) {
6      this.name = name;
7    }
8    // Getter:
9    public String getName() {
10     return name;
11   }
12 }
```

Following this, you write a class for students. It also has a name as an attribute, a constructor, and an accessor (Listing 9.21).

Listing 9.21: The class »Student«.

```
1  public class Student {
2
3    private String name;
4
5    public Student(String name) {
6      this.name = name;
7    }
8
9    public String getName() {
10     return name;
11   }
12 }
```

In Listing 9.22, you can see how to use both classes *Professor* and *Student* with a generic class. In this example, the generic base class is called *GenericRobot*.

Listing 9.22: The type-safe class »GenericRobot«.

```
1  public class GenericRobot<T> {
2
3    private T person;
4
5    public void setPerson(T person) {
6      this.person = person;
7    }
8
9    public T getPerson() {
10     return person;
11   }
12
13   public static void main(String[] arguments) {
14
15     GenericRobot<Professor> robert =
16       new GenericRobot<Professor>();
17     robert.setPerson(new Professor("Professor Roth"));
18
19     GenericRobot<Student> roland =
20       new GenericRobot<Student>();
21     roland.setPerson(new Student("Anna"));
22
23     System.out.println("Robert: I belong to " +
24       robert.getPerson().getName() + "!");
25
26     System.out.print("Roland: I belong to " +
27       roland.getPerson().getName() +"!");
28   }
29 }
```

The object *robert* is created in lines 19 and 20. Here, the program passes the correct target data type *professor* to the constructor of the class. In line 21, the program assigns the professor Roth to the object *robert*. The same happens with the object *roland*, which should belong exclusively to Anna.

Compare the program with the conventional, type-insecure version in the following listing. To achieve that the robot can be imprinted on different persons, the program defines the attribute *person* of type *Object*. This allows any objects to be passed, since Object is the base of all classes. In line 15, the program creates the robot object named *robert* and assigns it a Professor object titled »Professor Roth« in the next line.

To determine which person owns the robot, the program uses a typecast in line 21. A typecast is a type conversion. It is necessary because, by design, the method *getPerson()* returns only objects of type *Object*. Through the type cast, the program converts the return to the special type *Professor*. This goes well until a programmer makes a mistake, as in line 31. Here the program tries to determine from Roland a person of type *Professor*. The program is compiled by the compiler without a hitch, but ends up with an error during execution.

Listing 9.23: The type-insecure class »StandardRobot«.

```
1  public class StandardRobot {
```

```
2
3   private Object person;
4
5   public void setPerson(Object person) {
6     this.person = person;
7   }
8
9   public Object getPerson() {
10    return person;
11  }
12
13  public static void main(String[] arguments) {
14
15    StandardRobot robert = new StandardRobot();
16    robert.setPerson(new Professor("Professor Roth"));
17
18    StandardRobot roland = new StandardRobot();
19    roland.setPerson(new Student("Anna"));
20
21    Professor professor = (Professor) robert.getPerson();
22
23    System.out.println("Robert: I belong to " +
24      professor.getName() + "!");
25
26    Student student = (Student) roland.getPerson();
27
28    System.out.println(
29        "Roland: I belong to " + student.getName() + "!");
30
31    // Program error due to the assignment of an incorrect type:
32    professor = (Professor) roland.getPerson();
33
34    System.out.print("Roland: I belong to " + professor.getName() + "!");
35  }
36 }
```

This program error is completely eliminated in the type-safe variant of the program in Listing 9.22. This is because, thanks to the generic class, the Java compiler can detect and point out such errors as soon as the program is developed. This means that a faulty program cannot be compiled and delivered in the first place. Since such runtime errors can therefore no longer occur at all, generic classes are a significant advance achieved with Java 5. In addition, they avoid the labor-intensive and cumbersome type conversion. Therefore, development environments such as Eclipse are set to warn the programmer if he is not using generics where it would make sense.

9.7 Summary

Java has four different types of classes: concrete classes, abstract classes, interfaces, and generic classes (generics). Concrete classes could be called »normal« classes. Objects can be derived from this type of class using the keyword *extends*.

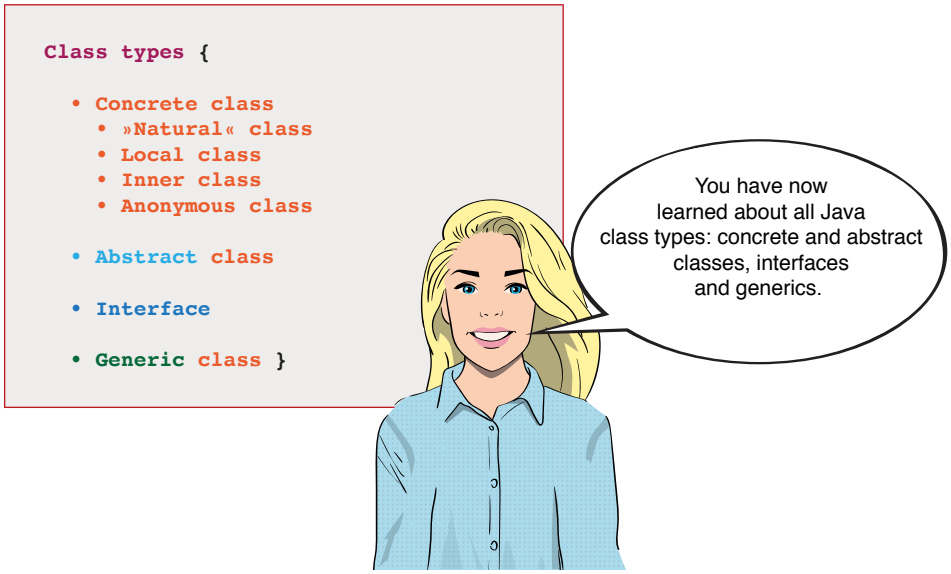


Figure 9.32: Anna summarizes the chapter from her point of view.

In contrast, you use interfaces and abstract classes for design reasons. Generic classes can be »normal« classes or interfaces. They are universal containers for all possible types of objects. The great advantage of generic classes is type safety. This means that the programmer can no longer make a mistake when assigning the objects, otherwise the program will not be compiled (Section 9.8, »Literature«).

9.8 Literature

Type safety: *https://de.wikipedia.org/wiki/Typsicherheit*
Oracle tutorials: *https://docs.oracle.com/javase/tutorial/java/TOC.html*
Java Object Tool: *https://openjdk.java.net/projects/code-tools/jol*

9.9 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/classes_and_objects

9.10 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge: *https://www.programmingcourse.net/courses/java_with_eclipse/classes_and_objects*

There you will also find the solutions to the exercises.

10 Enumerations

10.1 Introduction

You will encounter enumerations at every turn when programming. It is therefore a good thing that the Java programming language has had a data type called *enum* since version 5.

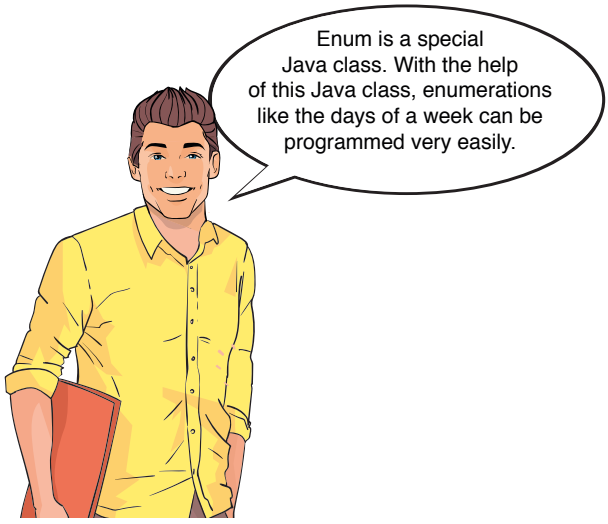


Figure 10.1: Lukas has hated enumerations since PE at school.

Enum is short for enumeration. The enumeration type is used to summarize collections of constants, for example the days of a week.

10.2 Overview

10.2.1 Purpose of Enums

The programming language C/C++, from which Java was strongly influenced, did not have a special data type for enumerations. And this probably explains why the first versions of Java did not provide a special data type for enumerations either. So Java programmers had no choice but to program each enumeration from scratch using its own class.

Over time, programmers published a whole series of so-called design patterns. These are particularly successful examples of Java classes, which should serve other programmers as a template. Using an example of days of the week, Listing 10.1 shows what an implementation of such an enum class looked like up to Java 1.4.

Listing 10.1: Example of an enum pattern up to Java 1.4.

```
1  //Examples/Enumerations
2
3  package programmingcourse;
4
5  class Weekday {
6    public static final Weekday MONDAY  = new Weekday("Monday");
7    public static final Weekday TUESDAY = new Weekday("Tuesday");
8    public static final Weekday WEDNESDAY = new Weekday("Wednesday");
9    public static final Weekday THURSDAY = new Weekday("Thursday");
10   public static final Weekday FRIDAY = new Weekday("Friday");
11   public static final Weekday SATURDAY = new Weekday("Saturday");
12   public static final Weekday SUNDAY = new Weekday("Sunday");
13
14   private String name;
15
16   //Constructor:
17   private Weekday( String name) {
18     this.name = name;
19   }
20
21   public String toString() {
22     return name;
23   }
24 }
```

In lines 6 to 12, you can see the individual elements of the enumeration. They are constants of type *string*. Each constant calls the constructor of the class *Weekday* in line 17 via the statement *new Weekday("...")*. The constructor is passed the name of the element as a parameter in the form of the data type *String* (i.e. as a string). The class *Weekday* stores this attribute in the object variable *name* of type *String*, which is on line 14. The class has one more method, *to String()*, which returns the name of an element.

Listing 10.2 shows how to use the Java class we developed ourselves. You can address each constant directly with a fixed value. As usual with *public static final* class variables (constants), calling an element always follows the scheme *Class.CONSTANT*.

Listing 10.2: Use of an enumeration pattern up to Java 1.4.

```
1  //Examples/Enumerations
2
3  package programmingcourse;
4
5  public class EnumerationPatternDemo {
6
7    public static void main(String[] args) {
8      System.out.println("Days of the week:");
9      System.out.println(Weekday.MONDAY);
10     System.out.println(Weekday.TUESDAY);
11     System.out.println(Weekday.WEDNESDAY);
12     System.out.println(Weekday.THURSDAY);
13     System.out.println(Weekday.FRIDAY);
14     System.out.println(Weekday.SATURDAY);
15     System.out.println(Weekday.SUNDAY);
16   }
17 }
```

Since each element gets its name passed individually by the constructor, the label and name of the element are arbitrarily different. A constant named *MONDAY* outputs the name as *Monday*. Thus, for the days of the week, the following program output results, which can be seen in Listing 10.3.

Listing 10.3: The program output of the enum pattern class.

```
Days of the week:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Programs with enums created before Java 5 and the introduction of enums were more or less different because there was no special language construct. Moreover, each programmer had to program a corresponding class by himself. In order to make enumerations in Java programs uniform and to simplify programming work, a special enumeration type has been an integral part of the Java programming language since version 5.

10.2.2 Definition And Declaration of Enums

Definition As a Separate Class

Enums can be defined in their own class or in a foreign class. Figure 10.2 shows how the definition looks like in a separate class. Unlike a standard Java class, the keyword *enum* introduces the class definition. This is followed by the name of the class.

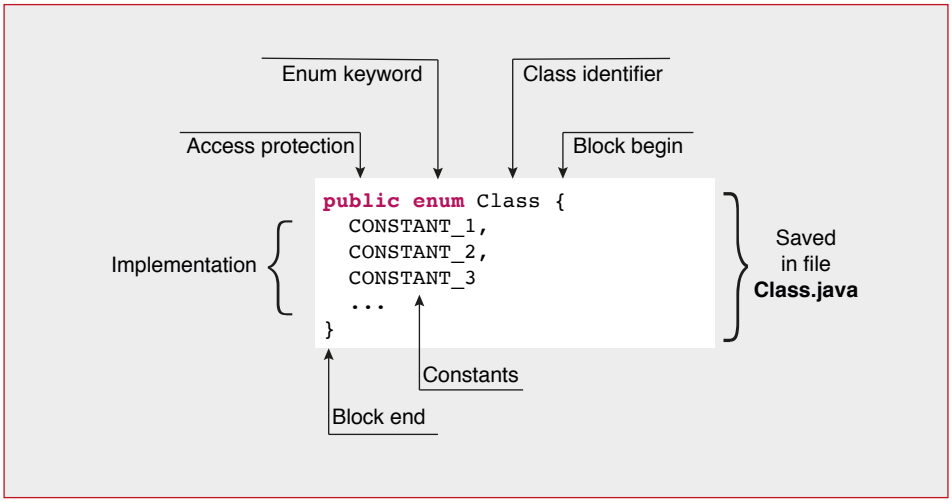


Figure 10.2: Definition of an enum as a separate class.

As with a standard Java class, the implementation is introduced with an opening curly bracket (block start) and ends with a closing curly bracket (block end). In between are constants that do not need to be declared. You simply write *MONDAY*, *TUESDAY*, *WEDNESDAY* etc. and the Java compiler translates this into a string accordingly. Note the notation of the elements. They are constants and should therefore always be written in capital letters.

Somewhat oddly, the name of the elements is coupled with their name. So if you define an element named *MONDAY*, the element will also output *MONDAY* as its name in uppercase. Compare that with the enumeration pattern class in Listing 10.1. Here, the name and spelling are independent. You will learn how to do this with the enum keyword in the next section.

Definition As a Inner Class

As with standard classes, you can define enums within a class if the enumeration is needed only locally in a class. In this case, a class surrounds the definition of the enum. The rest remains as it would be for a standalone class. Figure 10.3 shows how the definition looks as an inner class.

10.2.3 Usage of Enums

Enums contain constants that are already declared »ready to use«. Figure 10.4 shows how they are used. The example assigns the constant value on the right to a variable on the left.

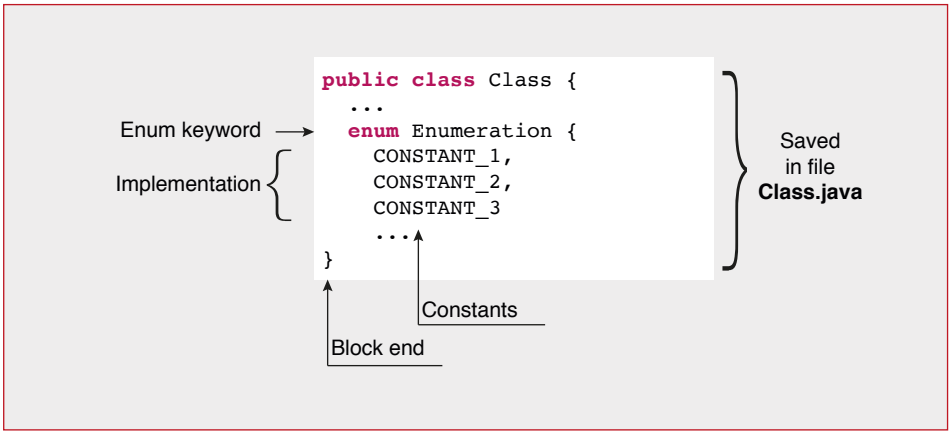


Figure 10.3: Definition of an enum as a inner class.

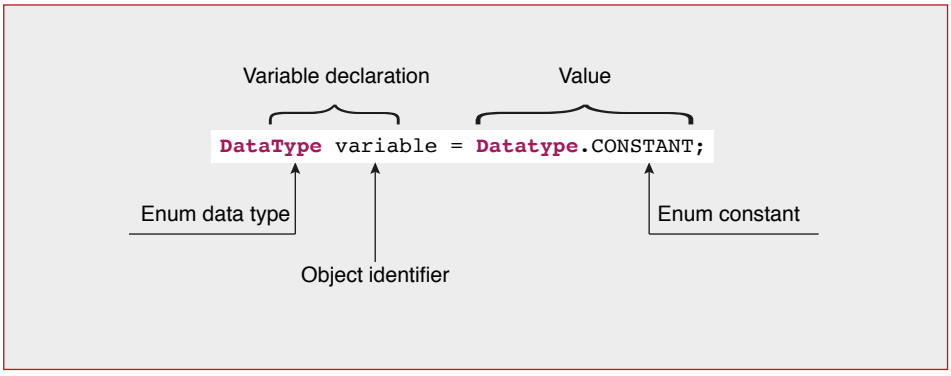


Figure 10.4: Usage of an enum.

10.3 Enum Classes

As mentioned at the beginning, the Java compiler creates a special enumeration class from the definition of an enumeration with keyword *enum*. This enumeration class is internally based on a base class called *Enum*. By using the *enum* keyword, you do not see this base class in the source code of your enumeration class.

So, using the class *Weekday* as an example, a class diagram would look like Figure 10.5. The figure is highly simplified for didactic reasons. The details would also go beyond the scope of this chapter. For a link to the detailed documentation of the class *Enum*, see Section 10.5, »Literature«.

The statement *enum Weekday* internally creates a derived enumeration class called *Weekday*, which is based on a base class *Enum*. All enumeration objects thus inherit from the internal class *Enum*. You will see in the rest of the chapter and in the exercise that follows that this information is important so that you can program enumeration classes with exactly the behavior you want.

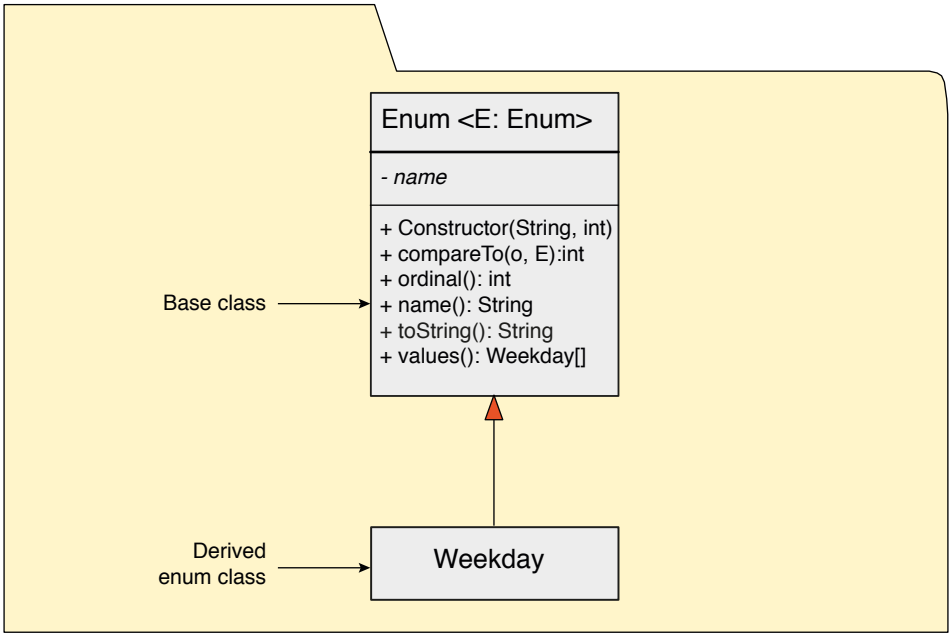


Figure 10.5: Enum is internally the base class of the enumeration classes.

10.3.1 Constructor

You never call this constructor directly. It is used internally to create the elements of an enumeration. This looks like the self-programmed enum class presented at the beginning (Listing 10.3):

```
Weekday(String value, int key);
```

However, the constructor here has two parameters. With the first, each element gets its name as a string. With the second, the class holds the position of the element within the list of enumerations. For example, *Monday* receives the value *0* as the first element – as is common in computer science.

10.3.2 Method »value()«

The method named *value()* returns all elements of an enumeration class. It is important to be able to call the values of an enum class one after the other in loops. This use case is covered in more detail later in this book section in chapter 15, »Loops«.

10.3.3 Separate Simple Enum Class

The definition using an independent class looks like Listing 10.4. Notice how shorter the class definition is compared to the example of Listing 10.1 shown at the beginning.

Listing 10.4: An example of a separate simple enum class.

```
1  package programmingcourse;
2
3  enum Weekday {
4    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
5  }
```

Listing 10.5 shows how to use the previously defined class. The program outputs each day of the week individually.

Listing 10.5: An example of using a separate simple enum class.

```
1  //Examples/Enumerations
2
3  package programmingcourse;
4
5  public class SeparateEnumClassDemo {
6
7    public static void main(String[] args) {
8      System.out.println("Days of the week:");
9      System.out.println(Weekday.MONDAY);
10     System.out.println(Weekday.TUESDAY);
11     System.out.println(Weekday.WEDNESDAY);
12     System.out.println(Weekday.THURSDAY);
13     System.out.println(Weekday.FRIDAY);
14     System.out.println(Weekday.SATURDAY);
15     System.out.println(Weekday.SUNDAY);
16   }
17 }
```

You then get a list where the program prints the names of the days of the week in uppercase (Listing 10.6). In this way, the enum type seems to be a step backwards compared to the flexibility of the self-programmed solution. However, this is only true if the names of the enumeration values also need to be output. In many cases, however, this is not necessary. The simple class definition of the Listing 10.4 is then completely sufficient.

Listing 10.6: The program output of the simple enum class.

```
The days of a week:
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

10.3.4 Separate Extended Enum Class

However, if you want to output the names of the individual elements in a notation that is different from the uppercase letters of the labels, you have to modify the enum class a bit. The example in Listing 10.7 shows how this works. It uses its own constructor (line 14) and overrides the *toString()* method of the base class, which outputs the name of an element, on line 19.

The technique of overwriting is discussed in detail in the third part of the book (Chapter 23, »Rules«, Section 23.7.2, »Method Overriding«). At this point, the only explanation needed is that, by overwriting, you replace the method of the base class. Thus, the new class achieves the same flexibility as a self-developed enumeration class, but with the advantage that class definition and access are standardized.

Listing 10.7: An example of an extended enum class.

```
1  //Examples/Enumerations
2
3  package programmingcourse;
4
5  enum Weekday {
6    MONDAY ("Monday"), TUESDAY ("Tuesday"),
7    WEDNESDAY ("Wednesday"), THURSDAY ("Thursday"),
8    FRIDAY ("Friday"), SATURDAY ("Saturday"),
9    SUNDAY ("Sunday");
10
11   private String name;
12
13   // Constructor with parameter name:
14   Weekday(String name){
15     this.name = name;
16   }
17
18   @Override
19   public String toString(){
20       return this.name;
21   }
22 }
```

The usage of this class is identical to the class before (Listing 10.8).

Listing 10.8: An example of using a separate extended enum class.

```
1  //Examples/Enumerations
2
3  package programmingcourse;
4
5  public class SeparateEnumClassDemo {
6
7    public static void main(String[] args) {
8      System.out.println("Days of the week:");
```

```
9       System.out.println(Weekday.MONDAY);
10      System.out.println(Weekday.TUESDAY);
11      System.out.println(Weekday.WEDNESDAY);
12      System.out.println(Weekday.THURSDAY);
13      System.out.println(Weekday.FRIDAY);
14      System.out.println(Weekday.SATURDAY);
15      System.out.println(Weekday.SUNDAY);
16    }
17  }
```

The program outputs the days of the week correctly written, because the names of the days of the week differ from the names of the elements in the class definition (Listing 10.9).

Listing 10.9: The program output of the extended enum class.

```
Days of the week:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

10.3.5 Inner Extended Enum Class

And here again is an example of a simple inner class (Listing 10.10).

Listing 10.10: An example of an inner extended enum class

```
1   //Examples/Enumerations
2
3   package programmingcourse;
4
5   public class InnerEnumClassDemo {
6
7     private enum Weekday {
8       MONDAY ("Monday"), TUESDAY ("Tuesday"),
9       WEDNESDAY ("Wednesday"), THURSDAY ("Thursday"),
10      FRIDAY ("Friday"), SATURDAY ("Saturday"), SUNDAY ("Sunday");
11
12      private String name;
13
14      // Constructor with parameter name:
15      Weekday(String name){
16        this.name = name;
17      }
18
19      @Override
20      public String toString(){
```

```
21          return this.name;
22        }
23      }
24
25      public static void main(String[] args) {
26        System.out.println("Days of the week:");
27        System.out.println(Weekday.MONDAY);
28        System.out.println(Weekday.TUESDAY);
29        System.out.println(Weekday.WEDNESDAY);
30        System.out.println(Weekday.THURSDAY);
31        System.out.println(Weekday.FRIDAY);
32        System.out.println(Weekday.SATURDAY);
33        System.out.println(Weekday.SUNDAY);
34      }
35    }
```

As you can see, with this class it is possible to define an inner class that correctly outputs the names of the elements. The output of the program is identical to Listing 10.9.

10.4 Summary

Enumerations are one of the most noticed news of Java 5. You saw at the beginning of this chapter a self-developed enumeration class based on a pattern written without a special Java keyword for enumerations. The keyword *enum*, available since Java 5, eliminates the need to program own enumeration classes from scratch. Another advantage of Java enums is that they increase the readability of Java programs and unify the source code.

As simple as enums are to use, their internal structure is complicated. Internally, the Java compiler translates an enum class into a special enumeration class that inherits from the internal class *Enum* and has various methods based on this base class. The behavior of an enumeration class can be customized to output arbitrary element names. Therefore, it is important to deal not only with the definition and use of enums, but also with the internals of this special class.

10.5 Literature

Enum: *https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html*

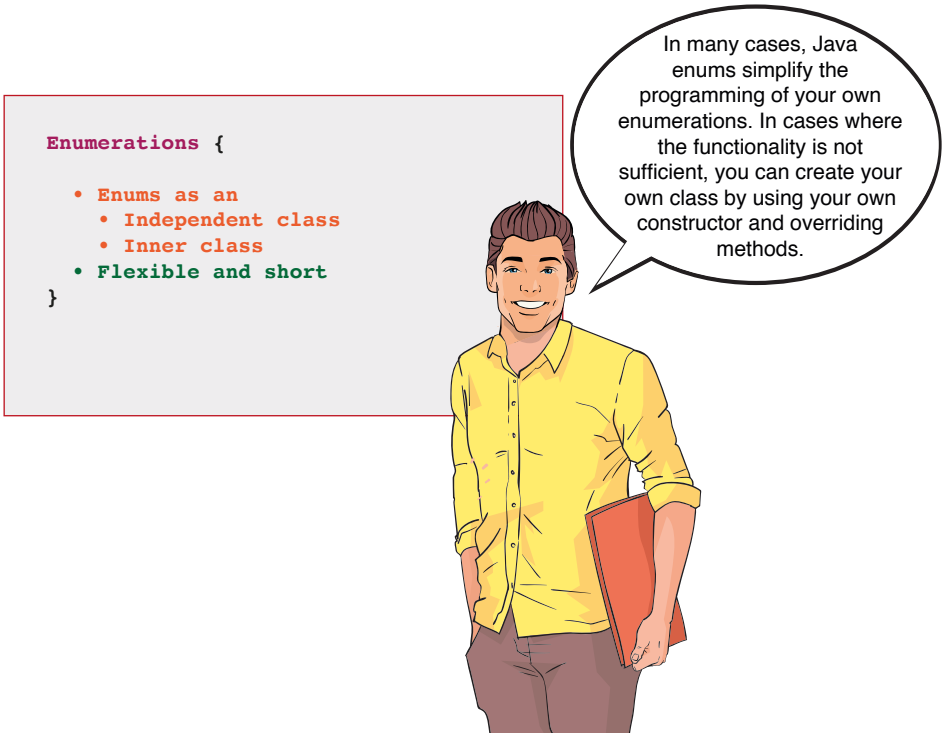


Figure 10.6 Enum is a Java special class for enumerations of any kind.

10.6 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/enumerations

10.7 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge: *https://www.programmingcourse.net/courses/java_with_eclipse/enumerations*

There you will also find the solutions to the exercises.

11 Arrays

11.1 Introduction

Arrays are simple data containers. They hold a number of variables of the *same* type. Arrays are needed, for example, to store and output simple tables.

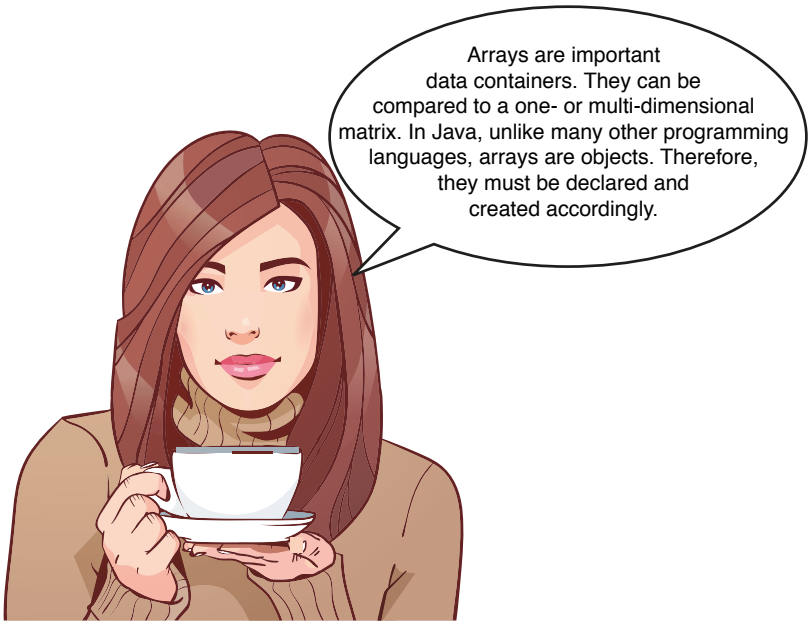


Figure 11.1: Julia will guide you through this chapter.

11.2 Overview

11.2.1 Purpose of Arrays

Arrays can be used to store sets of numbers or objects. Unlike some other programming languages, Java arrays are objects. Thus, they are created from a corresponding class. Arrays contain a fixed number of objects.

11.2.2 Types of Arrays

One-dimensional Arrays

There are one-dimensional or multi-dimensional arrays. One-dimensional arrays are comparable to enumeration types of the last chapter. They start with 0 and end at the element *n-1* (Figure 11.2).

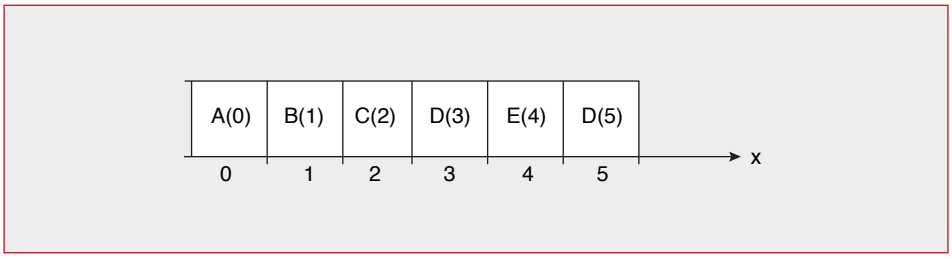


Figure 11.2: One-dimensional arrays behave like an enumeration.

Multidimensional Arrays

Multidimensional arrays are comparable to multidimensional matrices. They also start with 0 in each dimension and end at the element *n-1* (Figure 11.3).

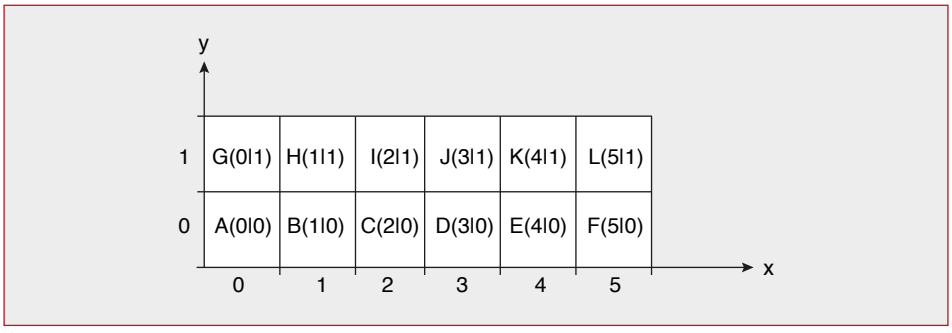


Figure 11.3: Multi-dimensional arrays behave like n-dimensional matrices.

11.2.3 Usage of Arrays

Declaration of a One-Dimensional Array

Like other variables, array variables must also be declared. This declaration is very interesting because it does not yet include the number of elements. For this reason, arrays are also called semi-dynamic. So, if you can determine how many elements the array should consist of before creating it within the program, you don't have to put in the number of elements until then. Note the pair of square brackets that follows the data type. It is the characteristic of a Java array. The pair of brackets is located to the right of the data type in Figure 11.4. In this case, the variable name does not contain brackets when the object is created.

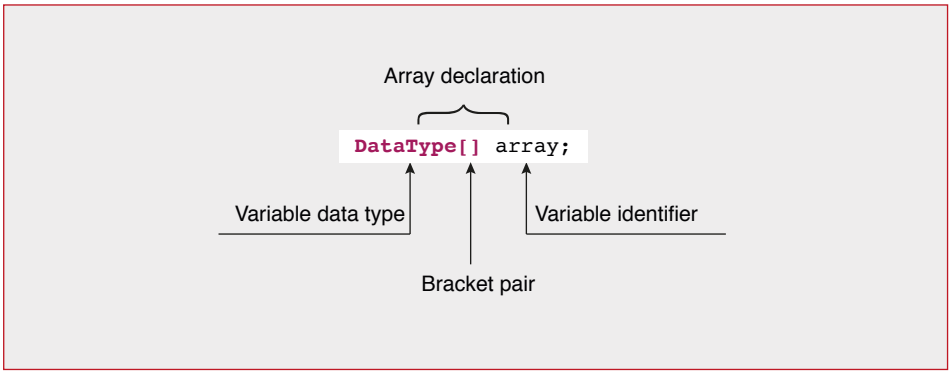


Figure 11.4: In this type of declaration, the pair of brackets is centered.

There is also another option to place the pair of brackets to the right of the variable (Figure 11.5). In this case, the variable name does contain brackets when the object is created (Listing 11.7). Which of the two possibilities one prefers is a matter of taste. Personally, I find the second way better, because it is clear from the variable name with brackets that it is an array.

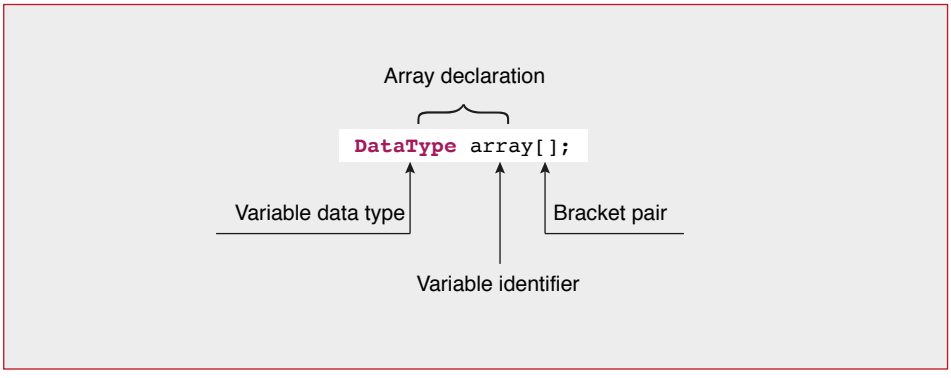


Figure 11.5: In this type of declaration, the pair of brackets is right-aligned.

Declaration of a Multidimensional Array

Multidimensional arrays are declared in the same way as one-dimensional arrays. For each dimension, another pair of brackets is needed. For a two-dimensional array, this looks like in Figure 11.6.

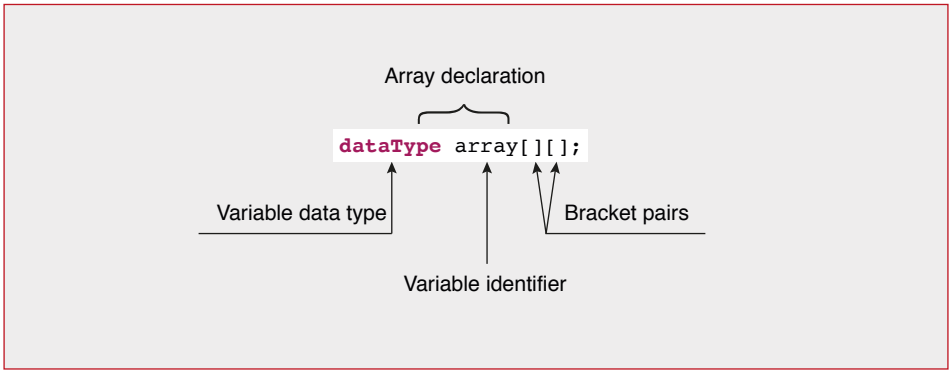


Figure 11.6: A multidimensional array has a pair of brackets for each dimension.

Creation of an Array With the New Operator

There are two methods available for creating an array. One creates an array classically like any other object with the new operator. This then looks like in Figure 11.7. Again, notice the pairs of brackets. The first pair of brackets in the name of the variable need to be named only if you have chosen the second type in the declaration, where the pair of brackets is part of the name (Figure 11.5).

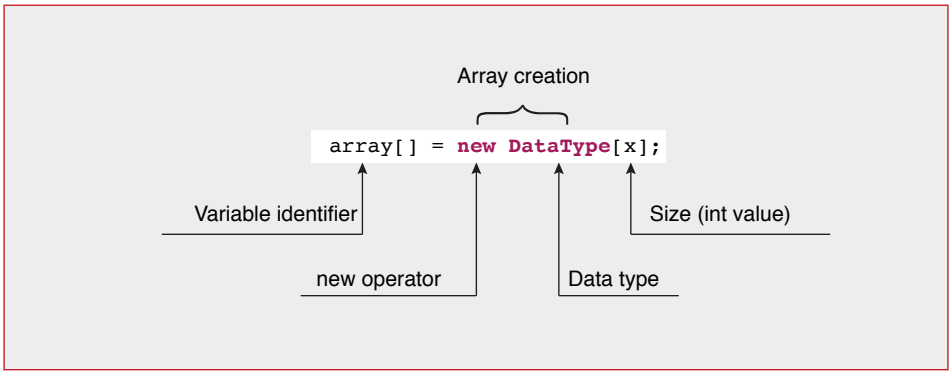


Figure 11.7: Creation of an array with the new operator.

The second pair of brackets specifies the number of elements of the array and thus the size of the array. This must be an integer value *int* (chapter 8, »Primitive Data Types«). You can find out the number of elements using the variable *length*, which each object of an array type has. To show how this type of array is used in a Java program, take a look at Listing 11.1.

Listing 11.1: An example of an array of person objects.

```
1  //Examples/Arrays/ArrayDemo
2
3  package programmingcourse;
4
5  public class ArrayDemo1 {
6
7    public static void main(String[] args) {
8      Person courseMember[] = new Person[6];
9      courseMember[0] =
10         new Person(Salutation.PROFESSOR, "Karsten", "Roth", false);
11     courseMember[1] =
12         new Person(Salutation.STUDENT, "Anna", "Seitz", true);
13     courseMember[2] =
14         new Person(Salutation.STUDENT, "Julia", "Lippert", true);
15     courseMember[3] =
16         new Person(Salutation.STUDENT, "Lukas", "Wittek", true);
17     courseMember[4] =
18         new Person(Salutation.STUDENT, "Florian", "Krause", true);
19     courseMember[5] =
20         new Person(Salutation.ROBOT, "Robert", "", false);
21     System.out.println("The programming course of Prof. Roth consists of"
22         + " " + courseMember.length + " persons:");
23     System.out.println("Professor " + courseMember[0].getFirstName() +
24       " " + courseMember[0].getLastName());
25     System.out.println(courseMember[1].getSalutation() +
26       " "+ courseMember[1].getFirstName());
27     System.out.println(courseMember[2].getSalutation() +
28         " "+ courseMember[2].getFirstName());
29     System.out.println(courseMember[3].getSalutation() +
30         " "+ courseMember[3].getFirstName());
31     System.out.println(courseMember[4].getSalutation() +
32         " "+ courseMember[4].getFirstName());
33     System.out.println(courseMember[5].getSalutation() +
34         " "+ courseMember[5].getFirstName());
35   }
36 }
```

The program creates an array with six elements for Professor Roth's programming course in line 8. To do this, line 8 shows a combination of the declaration and the creation of the array object. At this point, the array is still empty. Starting at line 9, the program creates an object of type *Person* for each member of the programming class and assigns it to an element of the array. From line 21 onwards, you will see how the program outputs the title of the course with the number of members. Here, the program accesses the *length* variable of the array. This is followed by other statements that output the name of the course members. The example program produces the following output:

```
Prof. Roth's programming course consists of 6 people:
Professor Karsten Roth
Student Anna
Student Julia
Student Lukas
```

```
Student Florian
Robot Robert
```

In both cases, you always need to know how big an array is at the time it is created. This is unfortunately a hindrance when programming, because often you want tables to grow dynamically.

Creation of an Array With a Shorthand Notation

Arrays can alternatively be created using a shorthand notation. Here, after the declaration, you specify in a block of statements which elements the array should consist of (Figure 11.8).

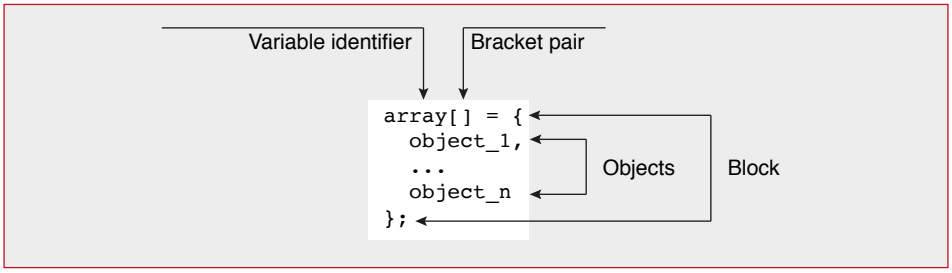


Figure 11.8: Creation of an array using a shorthand notation.

Transferred to the previous program example, this then looks like Listing 11.2. In line 8, the program declares the array in order to fill it with objects in lines 9 to 14. The notation is more compact than creating it with the new operator.

Listing 11.2: An example of creating an array using a shorthand notation.

```
1  //Examples/Arrays/ArrayDemo
2
3  package programmingcourse;
4
5  public class ArrayDemo2 {
6
7    public static void main(String[] args) {
8      Person courseMember[] = {
9         new Person(Salutation.PROFESSOR, "Karsten", "Roth", false),
10        new Person(Salutation.STUDENT, "Anna", "Seitz", true),
11        new Person(Salutation.STUDENT, "Julia", "Lippert", true),
12        new Person(Salutation.STUDENT, "Lukas", "Wittek", true),
13        new Person(Salutation.STUDENT, "Florian", "Krause", true),
14        new Person(Salutation.ROBOT, "Robert", "", false)
15     };
16     System.out.println("The programming course of Prof. Roth consists of"
17         + " " + courseMember.length + " persons:");
18     System.out.println("Professor " + courseMember[0].getFirstName() +
19       " " + courseMember[0].getLastName());
20     System.out.println(courseMember[1].getSalutation() +
```

```
21        " "+ courseMember[1].getFirstName());
22      System.out.println(courseMember[2].getSalutation() +
23          " "+ courseMember[2].getFirstName());
24      System.out.println(courseMember[3].getSalutation() +
25          " "+ courseMember[3].getFirstName());
26      System.out.println(courseMember[4].getSalutation() +
27          " "+ courseMember[4].getFirstName());
28      System.out.println(courseMember[5].getSalutation() +
29          " "+ courseMember[5].getFirstName());
30    }
31  }
```

The example program produces the same output as the program before:

```
Prof. Roth's programming course consists of 6 people:
Professor Karsten Roth
Student Anna
Student Julia
Student Lukas
Student Florian
Robot Robert
```

11.3 Summary

Arrays are very simple data containers. Unlike some other programming languages, an array variable in Java is an object. This object consists of further elements of any type. These elements are also objects.

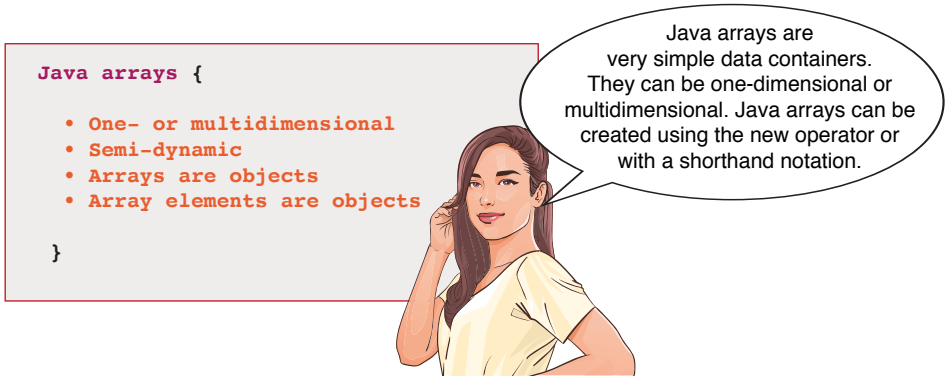


Figure 11.9: Julia is not enthusiastic about arrays because they are »only« semi-dynamic.

Because you do not have to specify how many elements the array will contain when you declare an array object, arrays are called semi-dynamic. After the declaration, when you create an array object, you must of course specify how many elements it should contain.

11.4 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/arrays

11.5 Exercises

When you have finished the tutorial, please go through the exercises to deepen your knowledge: *https://www.programmingcourse.net/courses/java_with_eclipse/arrays*

There you will also find the solutions to the exercises.

12 Methods

12.1 Introduction

So far you have learned how a Java program is structured, what data types are needed, and how to use variables. This is all part of the static side of Java programming.

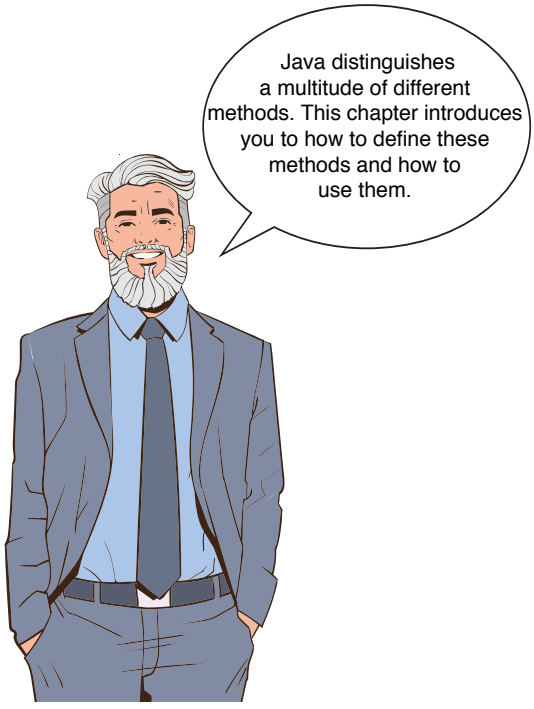


Figure 12.1: For Professor Roth, methods are the most exciting part of Java programming.

Methods belong to the dynamic part of programming: they start programs, create objects, perform calculations, return values and change them. In this chapter, you will learn which Java methods are used for this purpose, how they are structured and how to program them with Eclipse.

12.2 Overview

12.2.1 Method Purpose

Methods combine all the functions of a Java program. Without them, a Java program would be just dead matter. In Figure 12.2, you can see the interaction of various functions in the flow of the Java application *ProgramDemo*. In this diagram, the program *ProgramDemo* consists simplified only of the class *ProgramDemo*, a class *Person* and two objects.

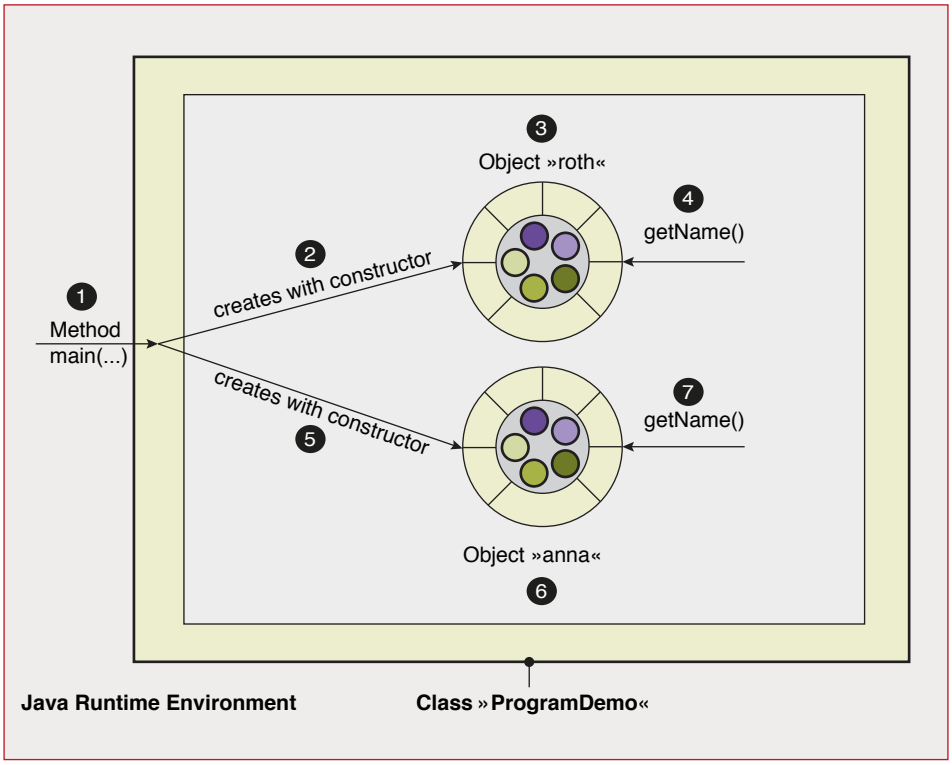


Figure 12.2: Simplified program flow of the application »ProgramDemo«.

The user starts the program via the Java Runtime Environment, calling the method *main()* of the class *ProgramDemo*. Then, using the constructors of the class *Person*, this method brings to life two person objects – the natural persons »Professor Roth« and »Student Anna«. From this point on, these objects can communicate with other objects and output their name and job title.

12.2.2 Method Types

Differences Between Methods

In Java, there are four fundamentally different Java methods (Table 12.1). In the first place is the constructor. Constructors are used to create objects from classes and assign their attributes. Therefore, every class from which objects can be created must have at least one constructor. Operations are most comparable to the functions of a classical programming language. They are used, for example, to start programs, calculate values, or output data to the screen.

Table 12.1: Java distinguishes four different types of methods.

Method type	Usage
Constructor	Creates an object
Operation	Classical function
Getter method	Accesses an attribute
Setter method	Changes an attribute

Together with operations, getter and setter methods form the interface of an object. Getter methods return the attributes of an object, while setter methods change its attributes. Think of the aforementioned method types as a membrane surrounding the nucleus to protect an object's attributes (Figure 12.3).

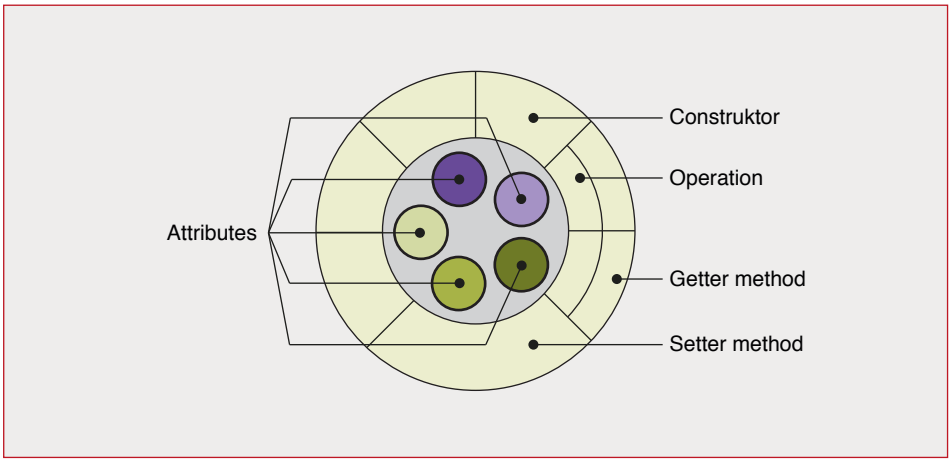


Figure 12.3: Methods form the interface of an object.

Technical Differences

From the technical point of view, operations, getter and setter methods can be divided into two types: Class methods and instance methods. Class methods, like class variables, are not bound to an object. They therefore exist from the time a class is loaded by the Java Runtime Environment until the end of a program. You do not need an object to call a class

method. The best known class method is the start method *main()* of a program. To execute this method, the program does not need an object.

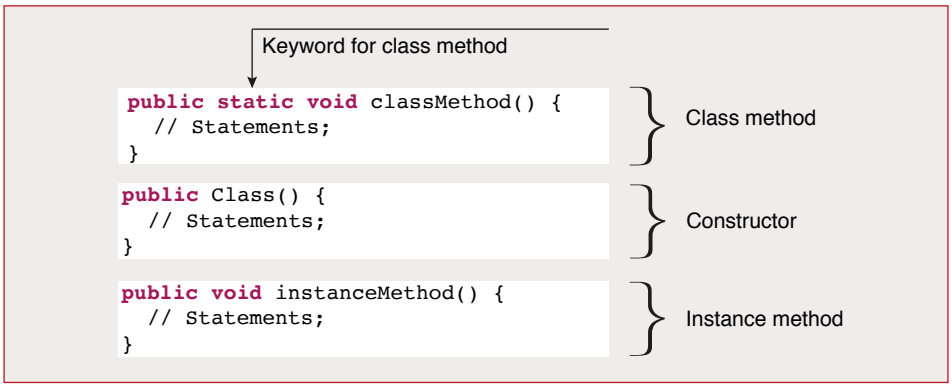


Figure 12.4: Comparison between a class method and an instance method.

The situation is different for instance methods. They are the classic object-oriented methods of a class. They form the interface of an object. Depending on how you design your classes, this protective shell is more or less permeable. The decision about how well you protect your objects is up to you. In Figure 12.4, you can see again the difference between a class method and an instance method. The constructor is in the middle as a special case.

12.2.3 Method Definition

Method Identifier

Unlike some other programming languages, every Java method must be part of a class. The definition[1] of a Java method is composed of the method header and its body (Figure 12.5).

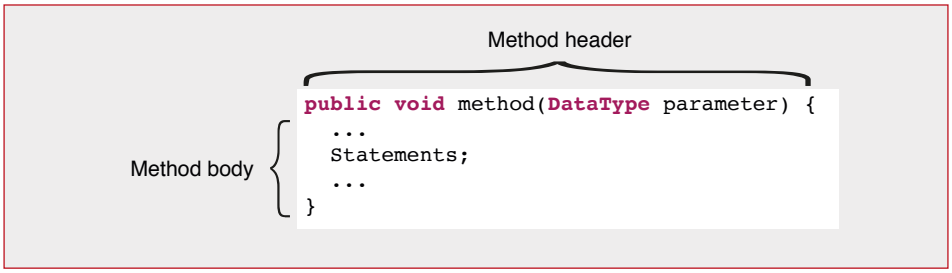


Figure 12.5: The structure of a method.

The header of a method consists of the specification of its visibility, the specification of its return value, and its signature (Figure 12.6). Signature means the name of the method (its name) and its parameter list[2]. The name of a Java method always starts with a lowercase

[1] Oracle sometimes uses the term »declaration« instead of »definition« at this point.
[2] Source: Oracle definition (see section 12.9, »Literature«).

letter, as with Java variables. Exceptions are constructors, which are always written the same way as the name of the class.

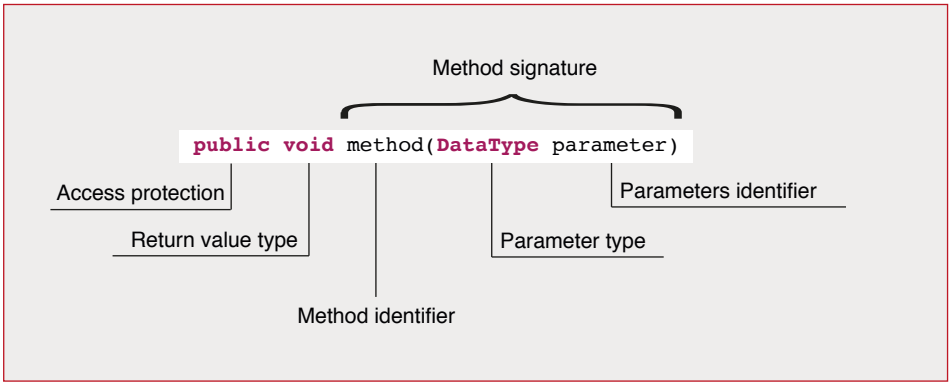


Figure 12.6: The header of a method.

In a class, there may be only *one* method with the same signature. Unfortunately, since the method signature includes only the name of the method and its parameter list, it is not possible in Java to define two methods with the same name that differ only in their return value, for example. Two query methods in one class, as seen in Listing 12.1, will result in a compiler error.

Listing 12.1: Multiple definitions with different return types are forbidden in Java.

```
1  public int getHeight() {
2    return height;
3  }
4
5  // Compiler error due to double definition:
6  public String getHeight() {
7    return Integer.toString(height);
8  }
```

Parameters

Parameters are used to pass data to a method. Even if a method should not have a parameter, as in Listing 12.1, it is still necessary in Java to append a pair of parentheses to the method label. The pair of parentheses indicates an empty parameter list. It is different from methods that pass parameters. This can be seen by comparing Figure 12.7.

In this figure, you can see a method without parameters in the first place above. The empty parameter list is indicated by a pair of parentheses. In the middle of the figure, you can see a method that has only one parameter. At the very bottom of the figure, you can see a method that takes multiple parameters.

A parameter list is always structured so that the type of the value follows first, and then the name. You may remember chapter 6, »Variables«, which already briefly introduced

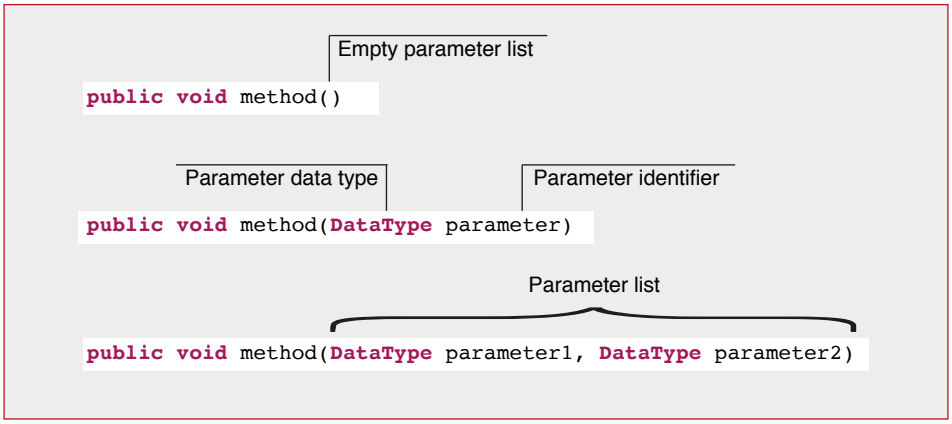


Figure 12.7: Comparison of methods without and with parameters.

parameters in section 6.4, »Parameters«: so you have to declare parameters just like normal variables. Separate multiple parameters with commas in the parameter list.

Return Values

All Java methods – except constructors – must specify in their method header whether you return a value, and if so, what type it is. So there are two cases:

- The method returns no value: return type *void*.
- The method returns a value: data type corresponding to the return value.

In Figure 12.8, you can see a comparison between a method without return value (left) and one with return value (right). In the figure on the left, you can see from the keyword *void* that this method does not return a value. In the method on the right, you can see the data type at this point. It must match the value that the method returns. The return introduces the method by the keyword *return*.

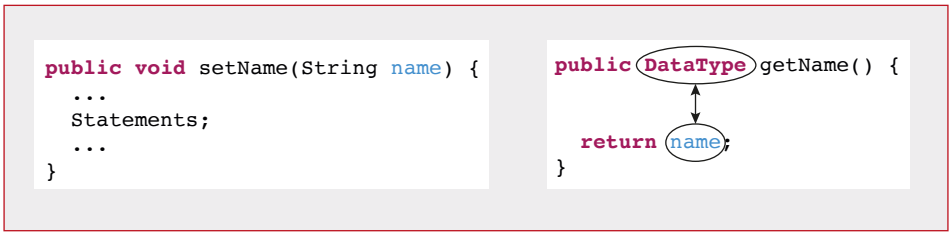


Figure 12.8: Comparison of methods without and with return value.

Access Protection

The type of the return value is preceded by the visibility of the method. To treat the visibility in more detail would go beyond the scope of this chapter. Only this much at this point: There are four levels to define the encapsulation of a method: *public*, *protected*, *default*, and *private*. Encapsulation is used to protect the method from access outside the object.

You can specify whether methods are visible only within an object, for example, or completely public throughout the program by cleverly choosing the encapsulation strength. You can learn more about this protection mechanism in part III, »Java Technology« (Chapter 23, »Rules«, section 23.4, »Access Protection«).

Method Body

The body of a method consists of statements and the actual implementation of the method. This implementation differs quite significantly from method type to method type. Figure 12.9 shows an overview of the four Java method types for this purpose. For constructors, the implementation consists of correctly assigning all the data necessary for the object and then transferring control of the program to other objects. One can hardly make a statement about the implementation of operations. Their implementation is as universal as their purpose. In the example of an operation *main()* in the figure, the method creates a new object called *julia*.

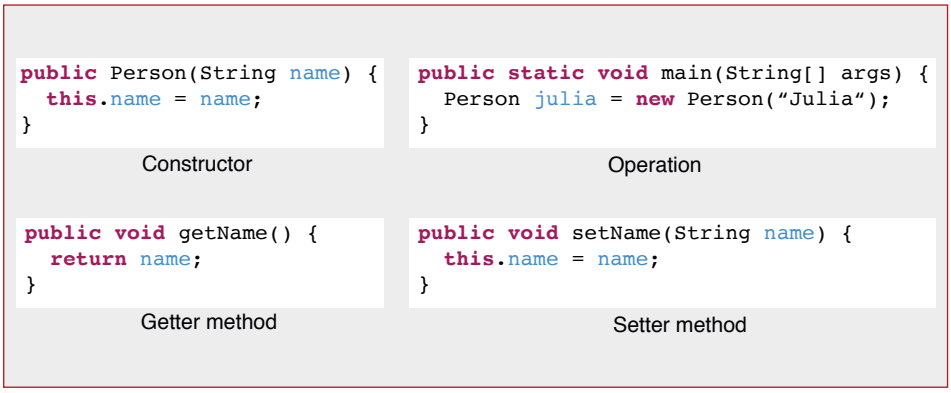


Figure 12.9: Comparison of the implementation of the four types of methods.

The implementation of query and change methods is mostly very simple. Both types of methods have only the sense of accessing attributes of an object. Therefore, for query methods, the implementation consists mainly of returning attribute values. Change methods take one or more parameters and thus overwrite the value of the attributes of an object.

So, now you've gotten an overview of how methods are defined. But how are the methods used? Again, you have to distinguish between the different types of methods.

12.2.4 Method Usage

Constructors

Constructors are very special Java methods also when they are used. They are used to create objects. Because of this special task, calling a constructor also differs from calling other methods. This is done using the new operator in the form of a statement (Figure 12.10).

In the left pane of the figure, you can see the definition of the constructor of the class *Calculator*. In the right pane, you can see the call to the constructor of this class. The call

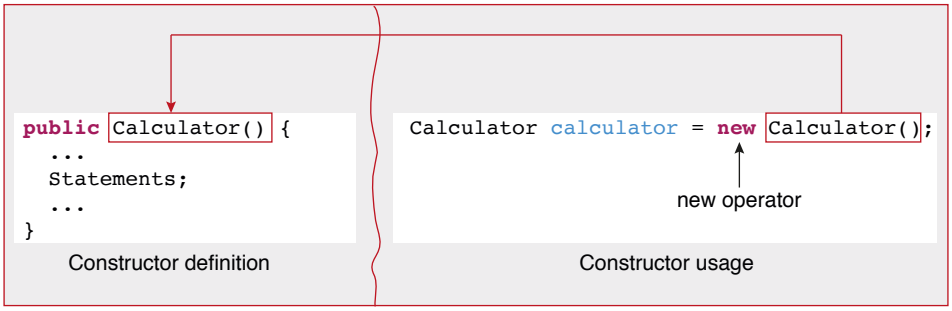


Figure 12.10: Usage of the constructor.

is introduced by the new operator, which is followed by the method. In this example, the constructor does not pass any parameter to the constructor.

Class Methods

Class methods do not require an object. Therefore, compared to other methods, the call is bound to a class. In the left part of Figure 12.11 the definition of a class method *calculateSum(...)* is listed. The class method has the keyword *static* as its flag. In the right part of the figure you can see how the call proceeds: First you specify the associated class, followed by the dot operator. Then the method is called according to its signature (name and parameters).

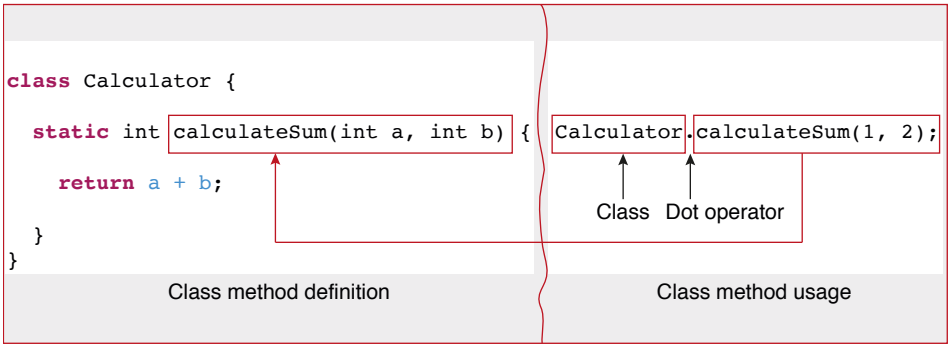


Figure 12.11: Usage of a class method.

Instance Methods

Instance methods need a concrete object with an identifier before you can call them. In the right part of Figure 12.12 you can see that the call is slightly different from that of a class method: First comes the name of the concrete object, which is followed by the dot operator. This is followed again by the method according to its signature (name and parameters).

Calling object methods using an anonymous object is unfamiliar to many programmers. It is very similar to calling a class method. Figure 12.13 shows how the call works: First, you create an anonymous object. This is simply an object to which you do not assign a variable.

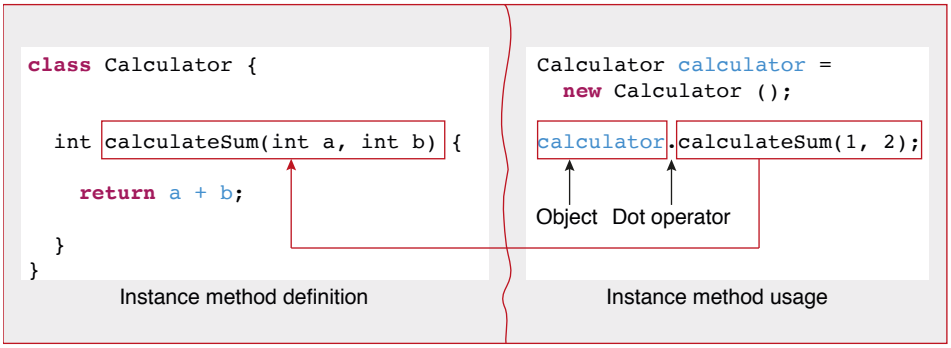


Figure 12.12: The use of an instance method of a concrete object.

Then follows the dot operator and the call of the method according to its signature (name and parameters).

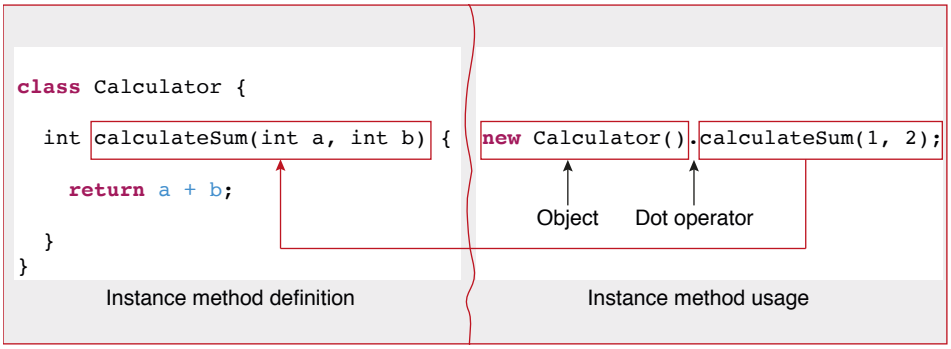


Figure 12.13: The use of an instance method of an anonymous object.

You may be wondering what to do with this nameless object. Using the method as an example, you can see that for some service in a program, it may not be necessary to create a named object if you only need the result of an operation. It is a matter of judgement whether a static method (class method) is not better suited for this.

This concludes the introduction to methods. In the further course of the chapter, you will now learn more about the four types of methods by means of program examples.

12.3 Constructors

The special method for creating objects is called a constructor (builder). It is used to create an object and possibly to assign defined values to it. There are standard constructors as well as constructors with and without parameters (Table 12.2).

Table 12.2: Java distinguishes three different types of constructors.

Constructor type	Explanation
Default constructor	Hidden constructor without parameters
Constructor without parameters	Self-defined constructor without parameters
Constructor with parameters	Self-defined constructor with parameters

12.3.1 Default Constructors

If you define a class without a constructor, the compiler automatically creates a default constructor in the translated class file when you compile the class. This is an empty constructor with no parameters. The compiler creates this default constructor because otherwise you could not create an object of this class. This constructor is very limited because it has no parameters to assign values to attributes. Listing 12.2 shows again the class *Person*. However, it has no *visible* constructor this time.

Listing 12.2: The class »Person« without a constructor.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Person {
6
7    String name;
8
9    public String getName() {
10     return name;
11   }
12
13   public void setName(String name) {
14     this.name = name;
15   }
16 }
```

Nevertheless, it is possible to call the constructor to create an object called *julia*. This happens in line 10 of the program *DefaultConstructorDemo* (Listing 12.3). To do this, the program calls the default hidden constructor and gives the object it creates the identifier *julia*. On line 12, the program calls the setter method *setName(String name)* to give the object called *julia* its »natural name«.

Listing 12.3: The class »Person« does not contain a visible constructor.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class DefaultConstructorDemo {
6
```

```
7    public static void main(String[] arguments) {
8
9      // Call of the standard constructor Person():
10     Person julia = new Person();
11
12     julia.setName("Julia");
13
14     System.out.println("Hello, I am " + julia.getName() + "!");
15   }
16 }
```

Then, on line 14, the program again accesses the variable *julia* to output the name of this object. Here, the program outputs what you see in Figure 12.14.

Figure 12.14: The program output of the person object »julia«.

12.3.2 Constructors Without Parameters

Sometimes you don't want it to be allowed to create an object without attributes. This can be the case, for example, when you want to forbid creating a person without a name. In this case, you need to prevent the default constructor from being called. This works, for example, by defining the default constructor as *private*. You can see this in line 8 of Listing 12.4.

Listing 12.4: The class »Person« with a private constructor.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Person {
6
7    String name;
8
```

```
9    // Private Constructor:
10   private Person() {}
11
12   public String getName() {
13     return name;
14   }
15
16   public void setName(String name) {
17     this.name = name;
18   }
19 }
```

Another option is to define a class that contains only constructors with parameters. In either case, compiling a class that attempts to call a constructor without parameters will result in an error (Listing 12.5).

Listing 12.5: A private constructor cannot be called from outside.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class PrivateConstructorDemo {
6
7    public static void main(String[] arguments) {
8      // The call of the private constructor generates the
9      // following compiler error in the Eclipse editor:
10     // "The constructor Person() is not visible"
11     Person julia = new Person();
12   }
13 }
```

12.3.3 Constructors With Parameters

In order to create individual objects, you need to write special constructors with various parameters. For example, the class *robot* could have a constructor with the parameter *name* (lines 10 to 13) and one with the parameters *address* and *name* (lines 15 to 19) (Listing 12.6).

Listing 12.6: The class »Robot« with two different constructors.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Robot {
6
7    private Salutation salutation;
8    private String name;
9
```

```
10    public Robot(String name) {
11      super();
12      this.name = name;
13    }
14
15    public Robot(Salutation salutation, String name) {
16      super();
17      this.salutation = salutation;
18      this.name = name;
19    }
20
21    public Salutation getSalutation() {
22      return salutation;
23    }
24
25    public String getName() {
26      return name;
27    }
28  }
```

It is useful to equip a class with a large number of such constructors for a wide variety of uses. The technique is called overloading methods. This technique is described in chapter 23, »Rules«, in section 23.7.1, »Method Overloading«.

If you want to create objects from the class *Robot*, you can pass only the parameter *name* in one case (line 10) and both parameters in the other case (line 15). Due to the fact that in the case of the robot object *roland* only one parameter is transferred, the attributes of this object are not fully initialized. This means that the object is not completely filled with meaningful values. In this case, it would make no sense to output the values. You can see the corresponding program in Listing 12.7.

Listing 12.7: This program creates two objects with different constructors.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class ParameterConstructorDemo {
6
7    public static void main(String[] arguments) {
8
9      // Call with parameter
10     Robot roland = new Robot("Roland");
11     System.out.println("Hello, I am " + roland.getName() + "!");
12
13     // Call with two parameters
14     Robot robert = new Robot(Salutation.ROBOT, "Robert");
15     System.out.println("Hello, I am a " +
16       robert.getSalutation() + " called " + robert.getName() + "!");
17   }
18 }
```

The program prints a greeting from the robot *roland* on line 11 and a greeting from the robot *robert* on lines 15 and 16 (Figure 12.15).

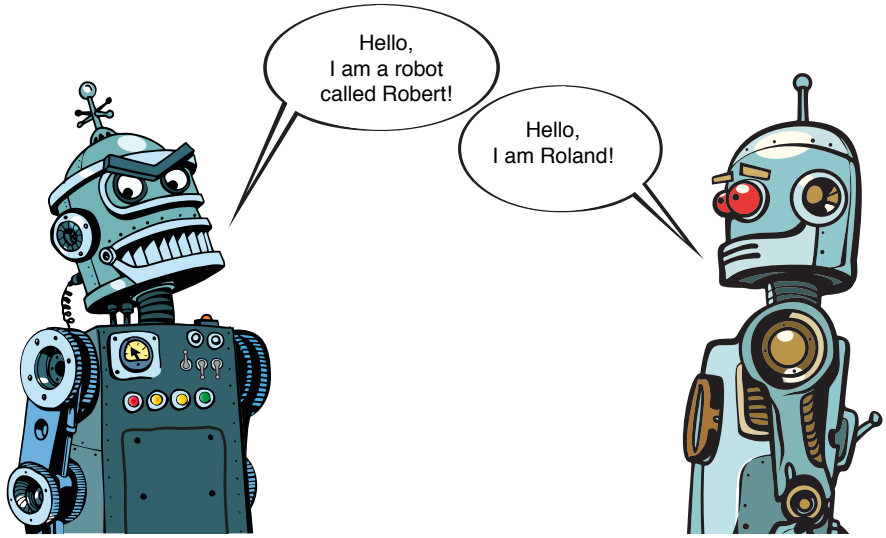


Figure 12.15: Program output of the program with two different constructors.

As mentioned at the beginning, constructors do not return concrete values, but they still must not be marked with *void* to distinguish them from »normal« methods. A definition like

```
public void Robot(String name) {
  ...
}
```

is interpreted as a normal method, although it starts with a capital letter. It has a completely different effect than the constructor:

```
public Robot(String name) {
  ...
}
```

If you define a class *robot* that contains only one method *robot()* of type *void*, it will execute without complaint, even though normal methods are lowercase in Java. However, when creating an object of class *robot*, the program does not call the *person()* method, but the default constructor of the same name. Thus, you may get a completely different program flow.

12.4 Destructors

Destructors (destroyers) of objects in the sense of C++ do not exist in Java. The automatic Java memory management automatically destroys objects in Java programs that are no

longer needed and frees their memory. Therefore destructors are superfluous. However, there is a method called *finalize()*, which is defined in the superclass *object*. It can be overridden and thus adapted to your own needs. The method overridden in a derived class like *Robot*, can be equipped with an individual behavior, as shown by lines 17 to 20 of the listing 12.8. The technique of overwriting is discussed in detail in the third part of the book (Chapter 23, »Rules«, section 23.7.2, »Method Overriding«).

Listing 12.8: The class »Robot« overrides the method »finalize()«.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Robot {
6
7    String name;
8
9    public Robot(String name) {
10     this.name = name;
11   }
12
13   public String getName() {
14     return name;
15   }
16
17   @Override
18   protected void finalize() {
19     System.out.print("and I come back to the\nend of the book!");
20   }
21 }
```

The *finalize()* method is part of the base class *object*, from which all Java classes derive. You should override this method of the base class *Object* when critical operations need to be done at the end of a program. The program *DestructorDemo* (Listing 12.9) shows how this works.

Listing 12.9: The class »DestructorDemo« shows how to use »finalize()«.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class DestructorDemo {
6
7    public static void main(String[] arguments) {
8
9      //Roland the robot says goodbye for the time being:
10     Robot roland = new Robot("Roland");
11     System.out.println("My name is " + roland.getName());
12     roland = null;
13     System.gc();
```

```
14     }
15 }
```

The program sets the value of the object *roland* to *null*. This signals the Java Runtime Environment that this object is no longer needed and can be deleted. To ensure that this is done immediately, the program calls the Java garbage collector of the Java Runtime Environment in line 13. Before deleting Roland, the Java Runtime Environment calls the overridden method *finalize()* to give the object a chance to perform final actions. For this reason, the object *roland* can still indicate that he will return at the end of this book (Figure 12.16).

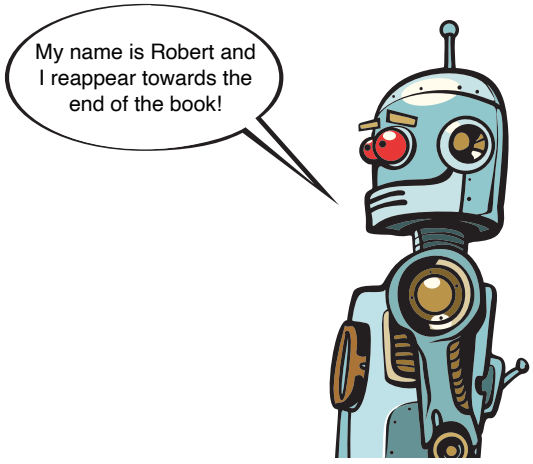


Figure 12.16: The program output of the program using the method »finalize()«.

12.5 Operations

Functions such as a calculation or starting a program are among the methods that are neither constructors nor query or change methods. Like the latter methods, they may or may not have return values. Unlike query or change methods, however, they do not (only) change attributes, but have more complex functions. One of the best-known examples of an operation is the start method of a Java program (Listing 12.10).

Listing 12.10: The method »main« starts a Java program.

```
1 public static void main(String[] arguments) {
2   instructions;
3 }
```

Another example of an operation is shown in Listing 12.11. The class method of the class *Calculator* performs a sum calculation.

Listing 12.11: The class »Calculator« with the operation »calculateSum(...)«.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Calculator {
6
7    public static int calculateSum(int a, int b) {
8      return a + b;
9    }
10 }
```

The program shows how to use the operation. By defining it as a class method, the program does not need an object to call the method.

Listing 12.12: The program »OperationDemo« shows how to call the class method.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class OperationDemo {
6
7    public static void main(String[] arguments) {
8      System.out.println("The sum of 1 + 2 = " +
9        Calculator.calculateSum(1, 2));
10   }
11 }
```

The program outputs the following result of the calculation.

```
The sum of 1 + 2 = 3
```

12.6 Getter Methods

12.6.1 Definition

If you want to get information from an object, you have to call an object method that returns a value. In general, these types of methods can be called »query methods«. However, most programmers call them »getter methods« or simply »getters« because they begin with the prefix »get«. If you are working with the Eclipse IDE, you should use the term getter methods because it is the common term with this development environment. Examples of getter methods have been plentiful so far in this book. Take another look at the class *Person* in Listing 12.13.

The class *Person* consists of five attributes for which there are corresponding getter methods. As mentioned before, each getter method starts with a »get«, hence its name. The only exception to this rule is the *isStudent()* method in line 53. Getter methods that return

a truth value always begin with »is«. The prefix is always followed by the name of the attribute in uppercase. The name of the getter is composed of the prefix and the name of the attribute. Classical getter methods do not have any parameters, because they only return the attribute of an object.

Listing 12.13: The getter methods return private attributes.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Person {
6
7    private Salutation salutation;
8    private String firstName;
9    private String lastName;
10   private String university;
11   private boolean student;
12
13   // 1. Constructor with two parameters:
14   public Person(String firstName, String lastName) {
15     this.firstName = firstName;
16     this.lastName = lastName;
17   }
18
19   // 2. Constructor with five parameters:
20   public Person(Salutation salutation,
21                 String firstName,
22                 String lastName,
23                 String university,
24                 boolean student) {
25     this.salutation = salutation;
26     this.firstName = firstName;
27     this.lastName = lastName;
28     this.university = university;
29     this.student = student;
30   }
31
32   public Salutation getSalutation() {
33     return salutation;
34   }
35
36   public String getFirstName() {
37     return firstName;
38   }
39
40   public String getLastName() {
41     return lastName;
42   }
43
44   public String getName() {
```

```
45      return firstName + " " + lastName;
46    }
47
48    public String getUniversity() {
49      return university;
50    }
51
52    public boolean isStudent() {
53      return student;
54    }
```

The getter methods are structured so that the method name is preceded by the data type of the return value. In line 32, the method *getSalutation()* returns the salutation of a person as an object named *salutation* of type *Salutation*. The methods use the *return* statement for this. It causes the method to return what follows the *return* keyword. In the case of the *getSalutation()* method, the expression is the attribute *salutation*. In the case of *getName()* in line 44, this is the value of the expression *firstName* + " " + *lastname*.

12.6.2 Usage

Getter methods are usually object methods. Therefore, an object is required to call a getter method. An example is shown in Listing 12.14. On line 8, the program creates the person object called *anna*. When it is created, the program passes five parameters.

Listing 12.14: The program »GetterDemo« shows how to use getters.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class GetterDemo {
6
7    public static void main(String[] arguments) {
8      Person anna =
9        new Person(Salutation.STUDENT, "Anna", "Seitz",
10         "University of Mainz", true);
11     System.out.println("Hello, I am " + anna.getFirstName() + ".");
12     System.out.print("I am " + anna.getSalutation());
13     System.out.println(" at the \n" + anna.getUniversity() + ".");
14   }
15 }
```

Figure 12.17 shows what the program outputs. The *student* attribute is suppressed for this program output, since the salutation already provides this information.

Figure 12.17: The program output of the person object »anna« with her getter methods.

12.7 Setter Methods

12.7.1 Definition

Methods that change the attributes of an object are the counterpart of getter methods. Because they change values, they are called »setter methods«, or »setters«. Setter methods write to an object – they change its state. The Eclipse IDE uses the term »getter methods« in all menus and dialogs, so you should remember the term well.

Unlike harmless getter methods, setter methods can do all sorts of damage to a program. For this reason, you should think carefully about which attributes you expose to the outside world and how you check passed parameters for correctness. Getter and setter methods usually form a pair for each attribute. The corresponding setter methods for the *Person* class can be found in Listing 12.15.

Listing 12.15: The setter methods of the class »Person«.

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class Person {
6
7    private Salutation salutation;
8    private String firstName;
```

```
9   private String lastName;
10  private String university;
11
12  public void setSalutation(Salutation salutation) {
13    this.salutation = salutation;
14  }
15
16  public void setFirstName(String firstName) {
17    this.firstName = firstName;
18  }
19
20  public void setLastName(String lastName) {
21    this.lastName = lastName;
22  }
23
24  public void setUniversity(String university) {
25    this.university = university;
```

The change methods do not return values, but take one or more values as parameters. Parameters are values that the program passes after the method name inside a pair of parentheses (lines 10, 14, 18, and 22). Since parameters must be declared, parameter passing also always follows the scheme *data type identifier*, as in a declaration.

In line 10 you can see that the class *Person* takes the parameter *address*. Then the program overwrites the internal value of the attribute of the same name with *this.salutation = salutation*. Because setter methods return *none* values, such methods are marked in Java with the keyword *void*.

The keyword expresses that the return value is *empty*. In this way, setter methods are easily distinguished from getter methods, which must define the data type of the return value at this point. They are equally easy to distinguish from constructors, which must not define anything at this point.

12.7.2 Usage

Listing 12.16 shows the use of setter methods using the example of the program *SetterDemo*. The program creates a new person object named *lukas* using the default constructor. In lines 9 through 11, the program calls the setter methods of the *person* class to pass the object the desired values for the *lukas* object.

Listing 12.16: The setter methods of the class »Person«

```
1  //Examples/Methods
2
3  package programmingcourse;
4
5  public class SetterDemo {
6
7    public static void main(String[] arguments) {
8      Person lukas = new Person();
```

```
9       lukas.setSalutation(Salutation.STUDENT);
10      lukas.setFirstName("Lukas");
11      lukas.setLastName("Wittek");
12      lukas.setUniversity("University of Mainz");
13      System.out.println("Hi, I am " + lukas.getFirstName() + ".");
14      System.out.print("I am " + lukas.getSalutation());
15      System.out.println(" at the \n" + lukas.getUniversity() + ".");
16    }
17  }
```

Figure 12.18 shows what the program outputs.

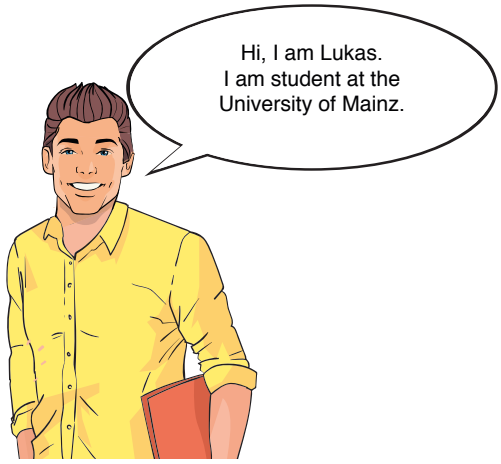


Figure 12.18: Program output of the person object »lukas« with its setter methods.

12.8 Summary

In Java, there are several types of methods: constructors create objects, getter and setter methods query and modify values. Operations are structured like functions of a classical programming language. With these methods one starts for example programs, accomplishes computations or makes database inquiries.

As with variables, Java distinguishes between instance and class methods. Instance methods require an object to be executed. Getter and setter methods are examples of instance methods. In contrast, no object is necessary to execute class methods. They are introduced by the keyword *static*. The best-known static method is *main(...)*, which is used to start a program.

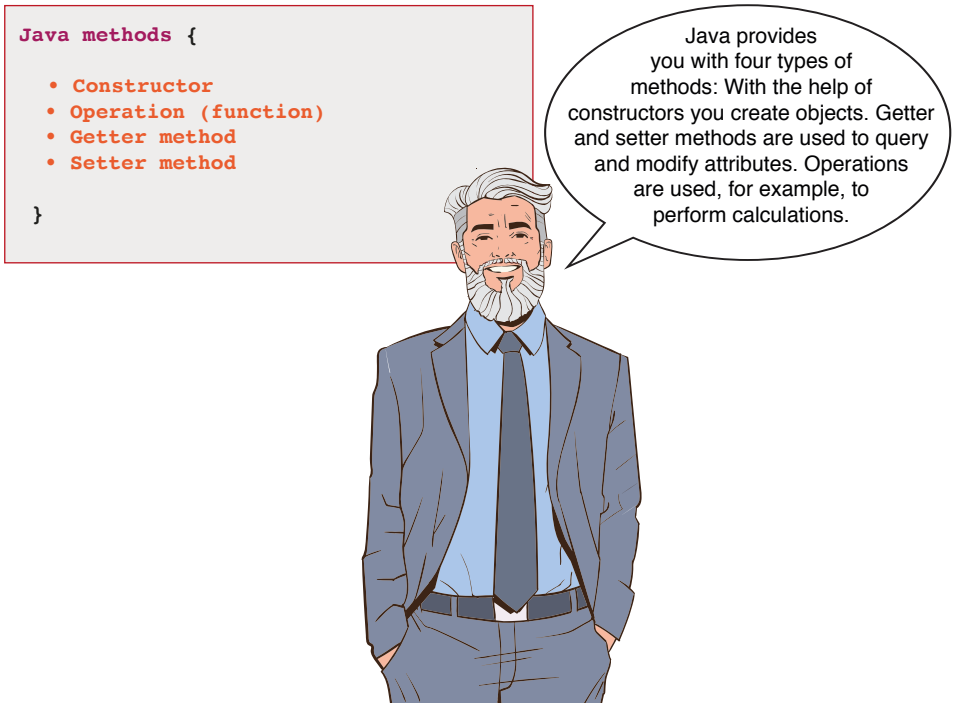


Figure 12.19: Professor Roth summarizes the characteristics of Java methods.

12.9 Literature

Methods: *https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html*

12.10 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/courses/java_with_eclipse/methods

12.11 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/courses/java_with_eclipse/methods
There you will also find the solutions to the exercises.

13 Operators

13.1 Introduction

Java operators are fixed Java keywords for special functions. So an operator is nothing else than the shorthand notation for a method. The keywords are very short, mostly only one digit. By operators variables, attributes and objects can be linked to expressions. The expressions then become complete statements. This chapter will show you again how this works by means of some examples.

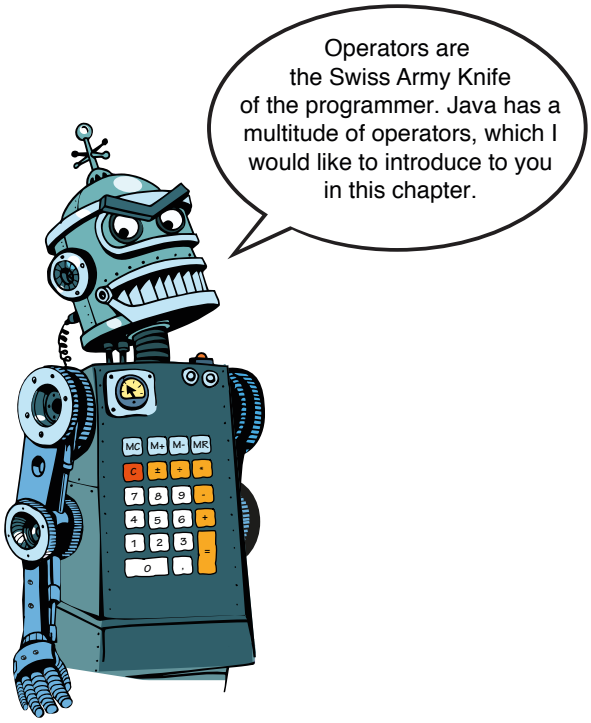


Figure 13.1: For Robert, operators are nothing more than a collection of tools.

13.2 Overview

13.2.1 Operator Types

Java provides operators for every use from arithmetic to comparisons and assignments, bit manipulations, type casts, to creating objects as well as object access (Table 13.1).

Table 13.1: This table shows an overview of the types of Java operators and their usage.

Operator	Usage
Arithmetic operators	Addition, subtraction, division, and multiplication
Relational operators	Comparison of expressions or types
Logical operators	Comparison of truth values
Bitwise operators	Bitwise manipulations
Assignment operators	Assignment of values
New operator	Creating an object
Cast operator	Temporary type conversion
Access operators	Access to anonymous classes and class components

13.2.2 Operator Precedence

Before we go into the operators in detail, it is important to know that operators are evaluated according to their precedence in expressions. Table 13.2 shows the order of precedence.

Table 13.2: The operators in this table are listed in order of precedence.

Operator	Precedence
Postfix	expr++ expr--
Unary	++expr –expr +expr -expr !
Multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< > <= >= instanceof
Equality	== !=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR	|
Logical AND	&&
Logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

The operators in the Table 13.2 are listed according to the precedence. The higher up an operator is in this table, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. All binary operators except assignment operators are evaluated from left to right; assignment operators are evaluated from right to left.

Operators that are in the same row in the table have the same precedence. If operators with the same precedence appear in the same expression, a rule must determine which operator is evaluated first. By rule is meant that expressions can be enclosed in parentheses, for example, to change the evaluation order.

13.3 Arithmetic Operators

The classical mathematical operators addition, subtraction, division and multiplication are also available in Java. In addition, there are also the operators that originate from C/C++ (Table 13.3).

Table 13.3: The table below lists all arithmetic Java operators.

Operator	Name	Example	Description
+	Unary plus operator	+i	Synonym for i
-	Unary minus operator	-i	Sign inversion of i
+	Additive operator	i + i	Addition or a String concatenation
-	Subtraction operator	i - i	Performs a subtraction
*	Multiplication operator	i * i	Performs a multiplication
/	Division operator	i / i	Performs a division
%	Remainder operator	i % i	Determines the division remainder
++	Pre-increment operator	j = ++i	1st step: i = i + 1, 2nd step: j = i
++	Post-increment operator	j = i++	1st step: j = i , 2nd step: i = i + 1
--	Pre-decrement operator	j = --i	1st step: i = i - 1, 2nd step: j = i
--	Post-decrement operator	j = i--	1st step: j = i , 2nd step: i = i - 1

13.3.1 Unary Plus Operator

The positive sign and the operator for an addition are identical characters in Java. In the case of the positive sign, the operator is called unary plus operator. Such a positive sign is always optional, so it does not have to be set separately.

The program in Listing 13.1 creates the person »Anna« with a height of +171 in line 9. In line 11, the program outputs the size of the person again:

```
Anna is 171 cm tall
```

Listing 13.1: This program shows a multiplication with two positive values.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class UnaryPlusOperatorDemo {
6
7    public static void main(String[] arguments) {
8
9      Person anna = new Person("Anna", +171);
10
11     System.out.println("Anna is " + anna.getHeight() + " cm tall");
12   }
13 }
```

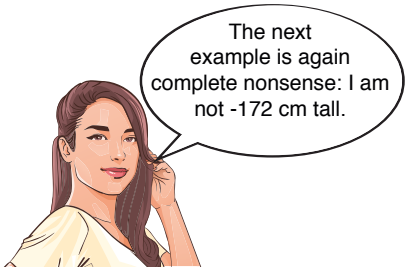


Figure 13.2: Julia always has something to complain about.

13.3.2 Unary Minus Operator

The negative prefix is analogous to the positive prefix: Both the negative sign and the subtraction operator are the same characters. But a negative sign does not cause a subtraction, but only a sign change. In addition, a negative prefix is not optional, unlike a positive one.

Listing 13.2: This program demonstrates a multiplication with two negative values.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class UnaryMinusOperatorDemo {
6
7    public static void main(String[] arguments) {
8
9      Person julia = new Person("Julia", -172);
10
11     System.out.println("Julia is " + julia.getHeight() + " cm tall");
12   }
13 }
```

The program shown in Listing 13.2 creates a person object with the id *julia* with a height of -172 in line 9 – which is nonsense, of course. In line 11 the program outputs the wrong height of the person *julia*:

```
Julia is -172 cm tall
```

13.3.3 Addition Operator

The addition operator, sometimes called additive operator, is used to add two variables. It is also able to join two strings together (Listing 13.3).

Listing 13.3: The addition operator adds two summands.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class AdditionOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9
10     int heightJulia =  172;
11
12     int totalSize = heightAnna + heightJulia;
13
14     System.out.println("Together we are " +
15       totalSize + " cm tall.");
16   }
17 }
```

In line 12 both variables are added and then in line 14 and 15 the output is concatenated with the same operator. You can see the program output in Figure 13.3.

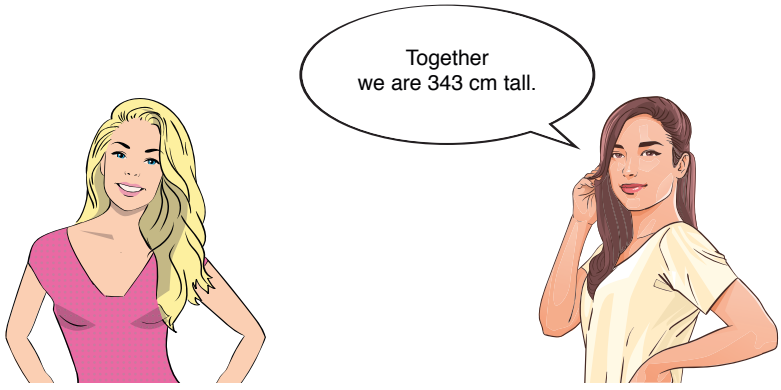


Figure 13.3: How tall are both female students together?

13.3.4 Subtraction Operator

With the subtraction operator, you perform a subtraction, here using the example of the size of the two persons Anna and Julia.

Listing 13.4: This program calculates the difference with the help of the subtraction operator.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class SubtractionOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9
10     int heightJulia =  172;
11
12     System.out.println("I am " +
13       (heightJulia - heightAnna) + " cm taller than Anna.");
14   }
15 }
```

The program outputs the following:

Figure 13.4: By how much is Julia taller than Anna?

13.3.5 Multiplication Operator

The product operator performs multiplication. The following example calculates the total capacity of male and female students of programming courses per semester:

Listing 13.5: This program multiplies two variables together.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class MultiplicationOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int studentPerProgrammingCourse = 4;
9      int numberOfProgrammingCourses = 4;
10     int capacity;
11     capacity = studentPerProgrammingCourse * numberOfProgrammingCourses;
12     System.out.println("Capacity = " + capacity + " students");
13   }
14 }
```

The output of the sample program is:

```
Capacity = 16 students
```

13.3.6 Division Operator

When using the division operator, note that Java programs store intermediate results of a division of integers as int values unless this is explicitly declared otherwise. Here is a short example that shows how errors occur and how to avoid them:

Listing 13.:6 Care should be taken when using the division operator with integers.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class DivisionOperatorDemo {
6
7    public static void main(String[] arguments) {
8      float result;
9      result = 1 / 5; // Error due to internal processing as integer
10     System.out.println("1. Result = " + result + " m (incorrect)");
11     result = 1F / 5F; // Correct by declaration as float
12     System.out.println("2. Result = " + result + " m (correct)");
13     result = (float) 1 / 5; // Correct by a cast to float
14     System.out.println("3. Result = " + result + " m (correct)");
15   }
16 }
```

The program outputs the following:

```
1. Result = 0.0 m (incorrect)
2. Result = 0.2 m (correct)
3. Result = 0.2 m (correct)
```

In the first case, you get the wrong result, because the program interprets both the numerator and the denominator as integer values (lines 9 and 10). In the second case, the program declares the numerator and denominator as float values, which is why the result is correct (lines 11 and 12).

In the third case, the program performs a conversion of the type of the division into a float value during the division. Therefore, this result is also correct (lines 12 and 14). Section 13.9, »Cast Operator«, presents such a type conversion again in more detail, while chapter 23, »Rules« discusses sources of errors in Java programs in detail.

13.3.7 Remainder Operator

The remainder operator, sometimes also called modulo operator, determines the remainder of an integer division. The following example 172 :171 = 1 results in a division remainder of 1, which the example also displays (Listing 13.7).

Listing 13.7: The remainder operator determines the division remainder.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class RemainderOperatorDemo {
6
7    public static void main(String[] arguments) {
8      Person anna = new Person("Anna", 171);
9      Person julia = new Person("Julia", 172);
10     int divisionRemainder = julia.getHeight() % anna.getHeight();
11     System.out.println("Division remainder: " + divisionRemainder);
12     divisionRemainder = anna.getHeight() % julia.getHeight();
13     System.out.println("Division remainder: " + divisionRemainder);
14   }
15 }
```

13.3.8 Pre-increment Operator

The following four operators are a legacy of C/C++. They combine assignments and calculations. The pre-increment operator first increments the value of the variable *heightAnna* and then assigns it to the variable *heightJulia*. The name of the pre-increment operator comes from Latin, where »prae« stands for »before« and »incrementare» for »to increase« or »to increment«.

Listing 13.8: The pre-increment operator first increments and then assigns.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class PreincrementOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna;
9      int heightJulia;
10     heightAnna = 171;
11     heightJulia = ++heightAnna;
12     System.out.println("The height of Anna is " + heightAnna + " cm");
13     System.out.println("The height of Julia is " + heightJulia + " cm");
14   }
15 }
```

The program outputs the following, which of course is wrong because the two female students are simply not the same size:

```
The height of Anna is 172 cm
The height of Julia is 172 cm
```

13.3.8.1 Post-increment Operator

The post increment operator does the opposite. It assigns the value of the variable *heightAnna* to the variable *heightJulia* in the first step and then increases the value of *heightAnna* in the second step. The name of the post-increment operator comes, of course, again from Latin: »post« means »after« and »incrementare« means »to increment«.

Listing 13.9: The post-increment operator first assigns and then increments.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class PostincrementOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna;
9      int heightJulia;
10     heightAnna = 171;
11     heightJulia = heightAnna++;
12     System.out.println("The height of Anna is " + heightAnna + " cm");
13     System.out.println("The height of Julia is " + heightJulia + " cm");
14   }
15 }
```

Also this time the program outputs completely wrong values:

```
The height of Anna is 172 cm
The height of Julia is 171 cm
```

13.3.8.2 Pre-decrement Operator

In the first step, the pre-decrement operator decrements the value of the variable *heightAnna* and then assigns it to the variable *heightJulia*. The operator's name comes again from Latin, where »prae« stands for »before« and »decrementare» for »to decrement«.

Listing 13.10: The pre-decrement operator first decrements and then assigns.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class PredecrementOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna;
9      int heightJulia;
10     heightAnna = 172;
11     heightJulia = --heightAnna;
12     System.out.println("The height of Anna is " + heightAnna + " cm");
13     System.out.println("The height of Julia is " + heightJulia + " cm");
14   }
15 }
```

The program again outputs a wrong value for Julia:

```
The height of Anna is 171 cm
The height of Julia is 171 cm
```

13.3.8.3 Post-decrement Operator

The post-decrement operator assigns the value to the variable *heightAnna* to *heightJulia* and then decrements *heightAnna*. The operator's name comes again from Latin, where »post« stands for »after« and »decrementare» for »to decrement«.

Listing 13.11: The post-decrement operator first assigns and then decrements.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class PostdecrementOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna;
9      int heightJulia;
10     heightAnna = 172;
11     heightJulia = heightAnna--;
12     System.out.println("The height of Anna is " + heightAnna + " cm");
13     System.out.println("The height of Julia is " + heightJulia + " cm");
14   }
15 }
```

The result of the program are different values and therefore correct.

```
The height of Anna is 171 cm
The height of Julia is 172 cm
```

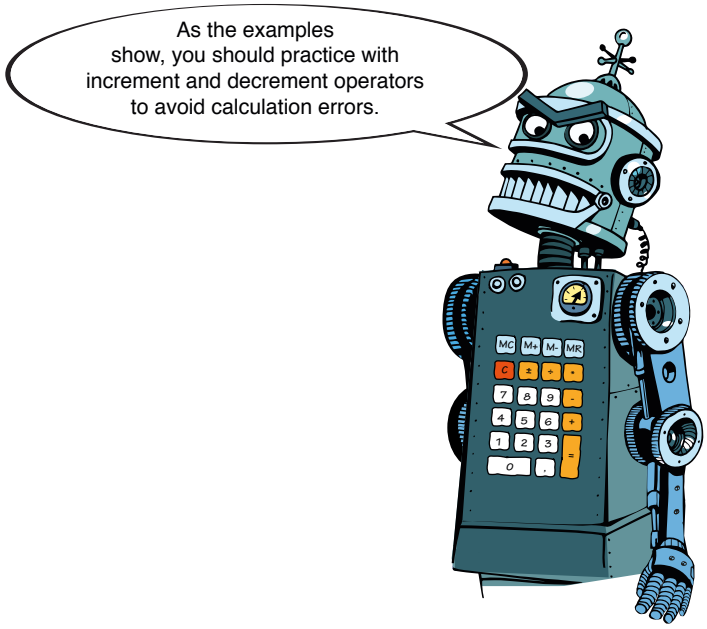


Figure 13.5: Incorrect use easily introduces calculation errors.

13.4 Relational Operators

Relational operators are used, as the name suggests, to compare expressions with each other. Again, first an overview of the available operators (Table 13.4).

Table 13.4: The table here shows all available relational Java operators.

Operator	Name	Example	Description
==	»Equal to« operator	i == j	Compares equal to
!=	»Not equal to« operator	i != j	Compares unequal to
<	»Less than« operator	i < j	Compares less than
<=	»Less than or equal to« operator	i <= j	Compares less or equal
>	»Greater than« operator	i > j	Compares greater than
>=	»Greater than or equal to« operator	i >= j	Compares greater than or equal to
instanceof	Type Comparison Operator	instanceof{object}	Compares the object type

13.4.1 »Equal to« Operator

The simplest operation is to check whether two expressions are identical. The result of the operation is a truth value. If two values are identical, the result is *true*, if not, *false*. Please note that the operator consists of *two* equal signs. Please do not confuse an assignment with only one equal sign (Section 13.7, »Assignment Operators«) with the equality operator.

Listing 13.12: This program compares whether Anna is as tall as Julia.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class EqualToOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Are Anna and Julia the same height? " +
11       (heightJulia == heightAnna));
12   }
13 }
```

The program generates the output:

```
Are Anna and Julia the same height? false
```

13.4.2 »Not Equal to« Operator

If you want to check if two values are not identical, you use the inequal operator.

Listing 13.13: This program compares whether Anna and Julia are different in height.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class NotEqualToOperator {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Aren't Anna and Julia the same height? " +
11       (heightJulia != heightAnna));
12   }
13 }
```

As expected, the program produces the reverse output this time:

```
Aren't Anna and Julia the same height? true
```

13.4.3 »Less Than« Operator

To find out if one expression or value is less than another, use the »less than« operator. Here is an example of how to use the operator:

Listing 13.14: This program compares if Anna is smaller than Julia.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class LessThanOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Anna smaller than Julia? " +
11       (heightAnna < heightJulia));
12   }
13 }
```

As expected, the program generates the following output this time as well:

```
Is Anna smaller than Julia? true
```

13.4.4 »Less Than or Equal to« Operator

The previous example looks different if you want to check whether the values are less than or equal to each other. So it is enough that the values are equal or less for the statement to be true (Listing 13.15).

Listing 13.15: This program checks whether Anna is smaller than or equal to Julia.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class LessThanOrEqualToOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Anna smaller or the same as Julia? " +
11       (heightAnna <= heightJulia));
12   }
13 }
```

The program generates the following output:

```
Is Anna smaller than or equal to Julia? true
```

13.4.5 »Greater Than« Operator

To find out if one expression or value is greater than another, you must use the »greater than« operator. The example (Listing 13.16) checks if Julia is taller than Anna.

Listing 13.16: This program checks if Julia is taller than Anna.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class GreaterThanOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Julia taller than Anna? " +
11       (heightJulia > heightAnna));
12   }
13 }
```

Since Julia is taller than Anna, this results in a true statement:

```
Is Julia taller than Anna? true
```

13.4.6 »Greater Than Or Equal To« Operator

If you want to check whether values are greater than or equal to each other, use the greater than or equal to operator. Here it is also sufficient that the values are either less than or equal to make the statement true (Listing 13.17).

Listing 13.17: This program checks if Julia is taller than or equal to Anna.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class GreaterThanOrEqualToOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Julia taller than or equal to Anna? " +
11       (heightJulia >= heightAnna));
12   }
13 }
```

Once again, a true statement results from this comparison:

```
Is Julia taller than or equal to Anna? true
```

13.4.7 Type Comparison Operator

The type comparison operator *instanceof* has a special role within the relational Java operators. It compares an object with a certain type. You can use it to check whether an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface. I will give here an example consisting of four classes. The first class is the base class *Being*. The class *Being* (Listing 13.18) is the base class for the class *Person* as well as for the class *Robot*. This means that both a person and a robot are instances of the class *Being* according to this definition.

Listing 13.18: The class »Being« is the base class for the classes »Person« and »Robot«.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Being {
6
7    private String name;
8
9    // Constructor with the parameter "name"
10   public Being(String name) {
11     this.name = name;
12   }
13
14   /**
15    * Returns the name of the object
16    * @return name
17    */
18   public String getName() {
19     return name;
20   }
21 }
```

The derived class *Person* (Listing 13.19) inherits all properties from the class *Being*. Each object of this class is both a being and a person. An instance of the class *Person* is the object *anna*.

Listing 13.19: The »Person« class is derived from the »Being« class.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class Person extends Being {
6
7    // Constructor with the parameter "name"
8    public Person(String name) {
9      super(name);
10   }
11 }
```

Analogous to the derived class *Person* , the class *Robot* (Listing 13.20) also inherits all properties from the class *Being*. Each object of this class is both a being and a robot. An instance of the class *Robot* is the object *robert*.

Listing 13.20: The »Robot« class is also derived from the »Being« class.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class Robot extends Being {
6
7    // Constructor with the parameter "name"
8    public Robot(String name) {
9      super(name);
10   }
11 }
```

The following program initially creates two objects named *anna* and named *robert*. Then, using the *instanceof* operator, the program queries whether *anna* is a being and, in the next line, whether *anna* is a person. Following this, the query for the object named *robert* is made.

Listing 13.21: This program demonstrates the use of the »instanceof« operator.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class TypeComparisonOperatorDemo {
6
7    public static void main(String[] arguments) {
8      Person anna = new Person("Anna"); // Object "anna" created
9      Robot robert = new Robot("Robert"); // Objekt "robert" created
10
11     // Is the object "anna" an instance of the class "Being":
12     if (anna instanceof Being)
13       System.out.println(anna.getName() + " is a being");
14
15     // Is the object "anna" an instance of the class "Person":
16     if (anna instanceof Person)
17       System.out.println(anna.getName() + " is a person");
18
19     // Is the object "robert" an instance of the class "Being":
20     if (robert instanceof Being)
21       System.out.println(robert.getName() + " is a being");
22
23     // Is the object "robert" an instance of the class "Robert":
24     if (robert instanceof Robot)
25       System.out.println(robert.getName() + " is a robot");
26   }
27 }
```

The program outputs the following:

```
Anna is a being
Anna is a person
Robert is a being
Robert is a robot
```

The use of the operator shows that the objects from type *Person* and *Robot* have only one thing in common: the base class *Being*. Both objects *person* and *robot* are instances of this base class. Hence the name of the operator *instanceof*, which you can use to analyze such inheritance relationships.

However, things are different for the derived classes. If you would try to compare a robot object with the class *Person*, the compiler would already report an error. It would signal with the help of the operator that the type of the object does not match the class. This means you can use the operator *instanceof* to find out while you are still developing a program that types of objects may not match.

Anna, of course, sees the example and the use of the operator quite differently, as you can see in the figure 13.6.

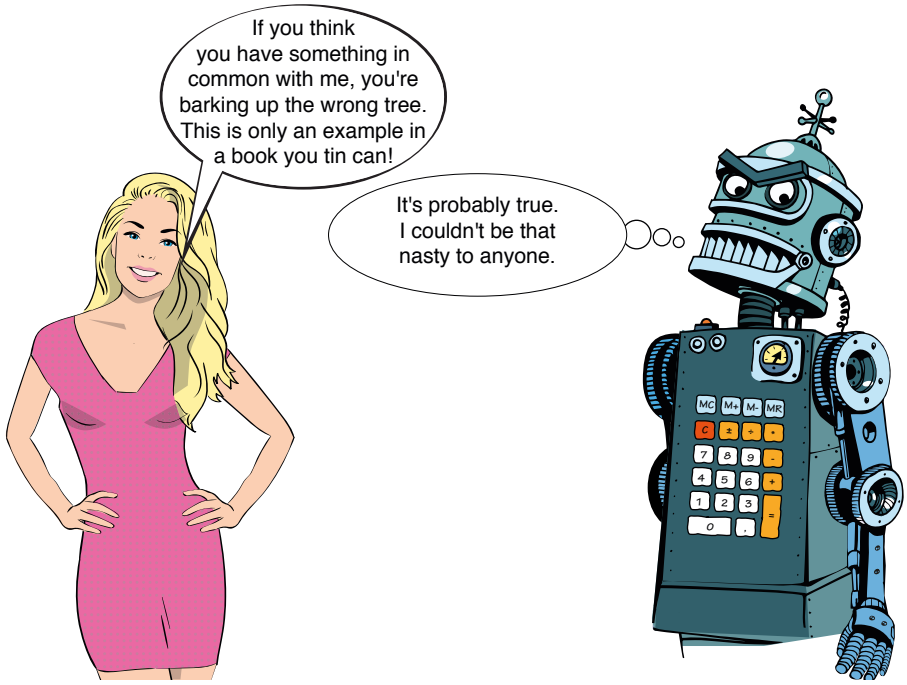


Figure 13.6: Anna sees nothing in common with Robert.

13.5 Logical Operators

Logical operators are also sometimes called conditional operators. You use these operators to compare truth values with each other. The following operators are available in Java:

Table 13.5: Logical operators.

Operator	Name	Example	Description
!	Logical complement operator	!i	Inverts the value of a boolean
&&	And operator	i && i	And-connection
||	Or operator	i || i	Or-connection
?:	Ternary operator	boolExpr ? a : b	Shorthand: if (boolExpr) then a else b

13.5.1 Logical Complement Operator

To reverse a boolean statement, one uses the logical complement operator. The example here compares two variables with each other. The result of this comparison is not true because Anna and Julia are different in height. To reverse the result, the program uses the logical complement operator. Therefore, the result is true again (Listing 13.22).

Listing 13.22: The non-operator inverts a statement.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class LogicalComplementOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Are Anna and Julia different in height? " +
11       (!(heightAnna == heightJulia)));
12   }
13 }
```

The program outputs the following:

```
Are Anna and Julia different in height? true
```

13.5.2 AND Operator

In the first step the following program tries to find out whether Anna is smaller than Julia. For this purpose the program compares the two variables *heightAnna* with *heightJulia*. The

result is a *true* statement. In the second step it checks again the variables *heightAnna* and *heightJulia* for equality. The result is a *false* statement. The AND operator combines a true and a false statement in such a way that the final result is *false*.

Listing 13.23: This program demonstrates an AND operation of two statements.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class AndOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Anna smaller than Julia AND"
11         + " are Anna and Julia the same height? " +
12         ((heightAnna < heightJulia) &&
13         (heightAnna == heightJulia)));
14   }
15 }
```

```
Is Anna smaller than Julia AND are Anna and Julia the same height? false
```

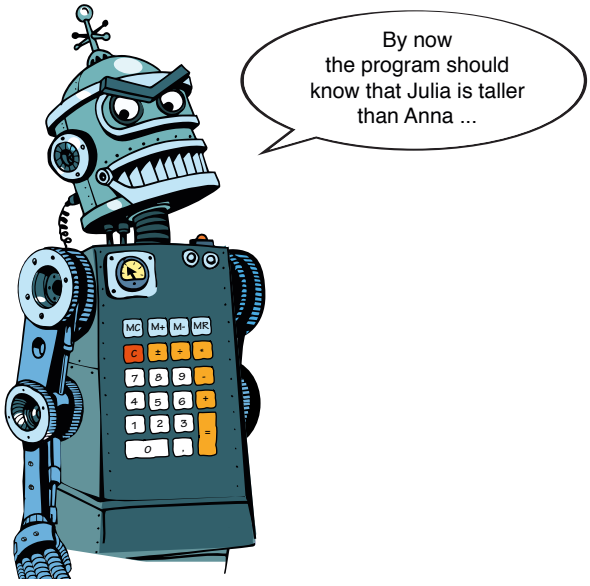


Figure 13.7: Robert hates logic errors in programs.

13.5.3 OR Operator

Again the same constellation, but this time with an OR operation. The result of the first expression is a *false* statement. The result of the second expression is a *true* statement. It is enough for one of the two statements to be true for an or-connection to return a true result. For this reason, the final result is *true*.

Listing 13.24: This program demonstrates an OR operation of two statements.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class OrOperatorDemo {
6
7    public static void main(String[] arguments) {
8      int heightAnna = 171;
9      int heightJulia = 172;
10     System.out.println("Is Anna smaller than Julia OR"
11         + " are Anna and Julia the same height? " +
12         ((heightAnna < heightJulia) ||
13         (heightAnna == heightJulia)));
14   }
15 }
```

```
Is Anna smaller than Julia OR are Anna and Julia the same height? true
```

13.5.4 Ternary Operator

The ternary operator is an extreme short form of a conditional statement. This kind of statements are introduced in detail in chapter 14, »Conditional Statements«. Because of its poor readability, this Java operator requires some programming practice. An example in listing 13.25 shows how to check a result using this operator.

Listing 13.25: The ternary operator replaces a conditional statement.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  class TernaryOperatorDemo {
6
7    static String determineStudentState(boolean student) {
8      String studentState =
9        ((student) ? " is a student" : " is not a student");
10     return studentState;
11   }
12
13   public static void main(String[] arguments) {
```

```
14      // Person "Anna", the object identifier is anna:
15      Person anna = new Person("Anna", true);
16      String studentState = determineStudentState(anna.isStudent());
17      System.out.println(anna.getName() + studentState);
18
19      // Person "Robert", the object identifier is robert:
20      Person robert = new Person("Robert", false);
21      studentState = determineStudentState(robert.isStudent());
22      System.out.println(robert.getName() + studentState);
23    }
24  }
```

The program outputs the following results in sequence:

```
Anna is a student
Robert is not a student
```

First the sample program checks if the variable *student* has the value *true*. If it does, it assigns the value *is a student* to the variable *studentState*, if not, it assigns the value *is not a student*. The long form of the program can be found in section 14, »Conditional Statements«, in section 14.3, »»If Then Else« Statement«.

13.6 Bitwise Operators

Bitwise operators are used to perform manipulations at the lowest level of a memory cell, the bit level (Table 13.6). They are not necessary for getting started in Java. Therefore, I will not go into detail about these operators here.

Table 13.6 This table shows an overview of the bitwise operators.

Operator	Name	Example	Description
|	Bitwise OR	i | i	Bitwise OR
&	Bitwise AND	i & i	Bitwise AND
^	Bitwise XOR	i ^ i	Bitwise exclusive OR (XOR)
>>	Signed right shift	i >> 2	Right shift with sign change
<<	Signed left shift	i << 2	Left shift with sign change
>>>	Unsigned right shift	i >>> 2	Right shift with positive sign change

13.7 Assignment Operators

Assignment operators, as their name implies, are used to assign values. Java essentially has the operators known from C/C++, which the table 13.7 summarizes. The assignment operators offer nothing new here, but only combine the previously known operators with an

assignment, so that you can save a line when writing a program. However, the readability of the program leaves much to be desired.

Table 13.7: This table shows an overview of the assignment operators.

Operator	Name	Example	Description
=	Simple assignment operator	i = 1	i gets the value 1
+=	Addition assignment operator	i += 1	i = i + 1
-=	Subtraction assignment operator	i -= 1	i = i - 1
*=	Product assignment operator	i *= 1	i = i * 1
/=	Division assignment operator	i /= 1	i = i / 1
%=	Ramainder assignment operator	i %= 1	i = i % 1
&=	AND assignment operator	i &= 1	i = i & 1
|=	OR assignment operator	i |= 1	i = i | 1
^=	Exclusive-OR assignment (XOR)	i ^= 1	i = i ^1
<<=	Left shift assignment	i <<= 1	i = i << 1
>>=	Right shift assignment	i >>= 1	i = i >> 1
>>>=	Right shift assignment with positive sign change	i >>>= 1	i = i >>> 1

13.8 New Operator

The new operator, which has already been used extensively in all examples, is used to create objects. I have listed it again here for the sake of completeness. It performs an operation that is used to create a new object and therefore belongs to the operators. The following listing defines a class for objects of type *person*.

Listing 13.26: The class »Person« defines a person.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Person {
6
7    String name;
8
9    public Person(String name) {
10     this.name = name;
11   }
12
13   public String getName() {
14     return name;
15   }
16 }
```

The following program again creates the person object named »Julia« and subsequently outputs the name of the student.

Listing 13.27: Creating an object with the new operator.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class NewOperatorDemo {
6
7    public static void main(String[] arguments) {
8      Person julia = new Person("Julia");
9      System.out.println("The person is called " + julia.getName());
10   }
11 }
```

13.9 Cast Operator

The conversion of a data type will be presented to you in more detail in chapter 23, »Rules«. At this point, there is a little anticipation of that chapter. The cast operator converts one variable into another. This is necessary in some situations, as the following example shows. The class *being* shall serve as a basis for the classes *human* and *robot* (Listing 13.28).

Listing 13.28: The base class »Being«.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Being {
6
7    private String name;
8
9    public Being(String name) {
10     super();
11     this.name = name;
12   }
13
14   public String getName() {
15     return name;
16   }
17 }
```

The class *Robot* automatically inherits the attribute *name* and the corresponding getter method from the base class *Being* (Listing 13.29).

Listing 13.29: The derived class »Robot«.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Robot extends Being {
6
7    public Robot(String name) {
8      super(name);
9    }
10 }
```

Like the class *Robot*, the class *Human* inherits the attribute *name* and the corresponding getter method. However, it extends the base class with the attributes *firstName*, *lastName*, and *student* (Listing 13.30).

Listing 13.30: The derived class »Human«.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Human extends Being {
6
7    String firstName;
8    String lastName;
9    boolean student;
10
11   public Human(String firstName, String lastName, boolean student) {
12     super(firstName + " " + lastName);
13     this.firstName = firstName;
14     this.lastName = lastName;
15     this.student = student;
16   }
17
18   public String getFirstName() {
19     return firstName;
20   }
21
22   public String getLastName() {
23     return lastName;
24   }
25
26   public boolean isStudent() {
27     return student;
28   }
29 }
```

In the program *CastOperatorDemo* all classes come together (Listing 13.31). To manage machine beings like »Robert« and humans like »Anna« together in our programming course, the program creates an array of objects of type *Being* (line 9). In the next two lines,

the program creates an object of type *Human* (line 11) and one of type *Robot* (line 12) and assigns both to the array.

Listing 13.31: A type conversion from »Being« to »Human«

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class CastOperatorDemo {
6
7    public static void main(String[] arguments) {
8      Being courseMember[] = new Being[2];
9
10     courseMember[0] = new Human("Anna", "Seitz", true);
11     courseMember[1] = new Robot("Robert");
12
13     System.out.println("The student is called "
14       + ((Human) courseMember[0]).getFirstName());
15     System.out.println("Is Anna a student: "
16       + ((Human) courseMember[0]).isStudent());
17     System.out.println("The robot is called "
18       + courseMember[1].getName());
19   }
20 }
```

However, an object of type *being* does not have methods *getFirstName()*, *getLastName()*, and *isStudent()*. To access these methods for outputting the student's first name and status, the program performs a type conversion using the cast operator on lines 15 and 17. There is nothing illegitimate about this, since the object is actually of type *Human*. Therefore, the access does not cause any compiler or runtime error.

The program outputs the following results:

```
The student is called Anna
Is Anna a student: true
The robot is called Robert
```

13.10 Access Operators

13.10.1 Dot Operator

The dot operator, which has been used several times in this book, can be used to access variables and methods of an object as well as class variables and methods. To explain the use of this operator again with an example, let me first introduce you to the enumeration class *Salutation*. It contains two constants: MISSES and MISTER (Listing 13.32).

Listing 13.32: The enumeration class »Salutation« with two constants.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public enum Salutation {
6    MISSES("Mrs."),
7    MISTER("Mr.");
8
9    private String salutation;
10
11   Salutation(String salutation) {
12     this.salutation = salutation;
13   }
14
15   @Override
16   public String toString() {
17     return salutation;
18   }
19 }
```

The class *Person* has a constructor that can be used to set the name and salutation of the person (Listing 13.33).

Listing 13.33: With the help of this class, person objects can be created with a salutation.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class Person {
6
7    private String name;
8    private Salutation salutation;
9
10   public Person(String name, Salutation salutation) {
11     this.name = name;
12     this.salutation = salutation;
13   }
14
15   public String getName() {
16     return name;
17   }
18
19   public Salutation getSalutation() {
20     return salutation;
21   }
22 }
```

The demo program (Listing 13.34) creates an object of the type *Person*. Here the constructor passes the name and salutation of the person (line 8). In line 9 you can see how the dot operator is used to call the getter methods. The syntax is as follows: To the left of the dot

operator, you specify the object or class. To the right of the dot operator is the variable or method you want to access.

Listing 13.34: This program shows the use of the dot operator.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class DotOperatorDemo {
6
7    public static void main(String[] args) {
8      Person roth = new Person("Karsten Roth", Salutation.MISTER);
9      System.out.println(roth.getName() + " is " + roth.getSalutation());
10   }
11 }
```

13.10.2 Lambda Operator

Lambda expressions are a shorthand way to create anonymous classes. To use them, a lambda operator is required. The operator is explained in chapter »Conditional Statements«, section 14.5.4 in detail using a conditional statement. Here is just a quick example showing the correlation between the size of a person and their name (Listing 13.35).

Listing 13.35: The lambda operator replaces an anonymous class.

```
1  //Examples/Operators
2
3  package programmingcourse;
4
5  public class LambdaOperatorDemo {
6
7    static String determinePerson(int height) {
8      return switch(height) {
9        case 171 -> "Anna";
10       default -> "No idea";
11     };
12   }
13
14   public static void main(String[] args) {
15     System.out.print("171 cm = " + determinePerson(171) + ", ");
16     System.out.print("220 cm = " + determinePerson(220));
17   }
18 }
```

The program outputs the following result:

```
171 cm = Anna, 220 cm = No idea
```

13.11 Summary

Java has arithmetic, relational, logical and assignment operators. Furthermore, there are operators for bit manipulation, for creating objects and for type conversion. The access operators complete the spectrum of operators provided by the Java programming language.

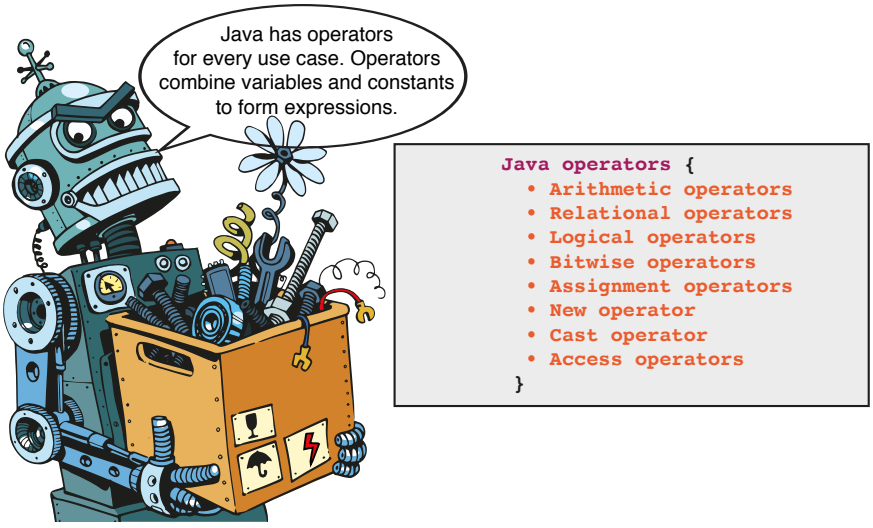


Figure 13.8: For Robert, operators are like a toolbox.

13.12 Literature

»Operators«: *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html*

13.13 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/operators

13.14 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/operators
There you will also find the solutions to the exercises.

14 Conditional Statements

14.1 Introduction

In the Java programs of this book so far so-called conditional statements were missing. Conditional statements are instructions for handling decisions. Through these statements, the program makes the decision to execute either one or the other part of a program. Conditional statements control the program flow. How this works and what different types of conditional statements there are in Java is presented in this chapter.

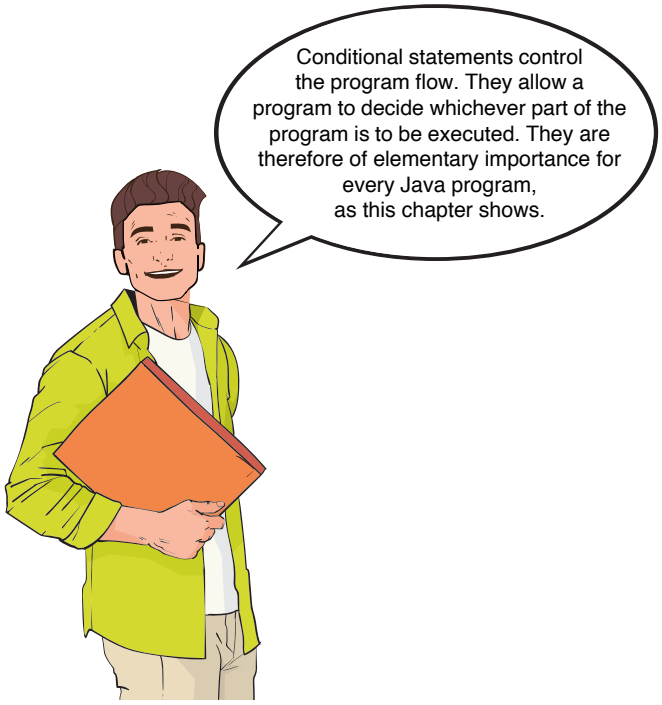


Figure 14.1: Florian hates control, but he finds conditional statements really cool.

14.2 Overview

Conditional statements are used to control the program flow. They therefore belong – like loops – to the control flow statements of a program. You can see an example of a flow chart with a conditional in figure 14.2. Here, at the beginning, where the green diamond is located, it is to be decided whether the person is enrolled at a university. The program distinguishes between two cases in this example: The person is a student, and the person is not a student. In each case, the program should be able to distinguish whether the person is enrolled or not.

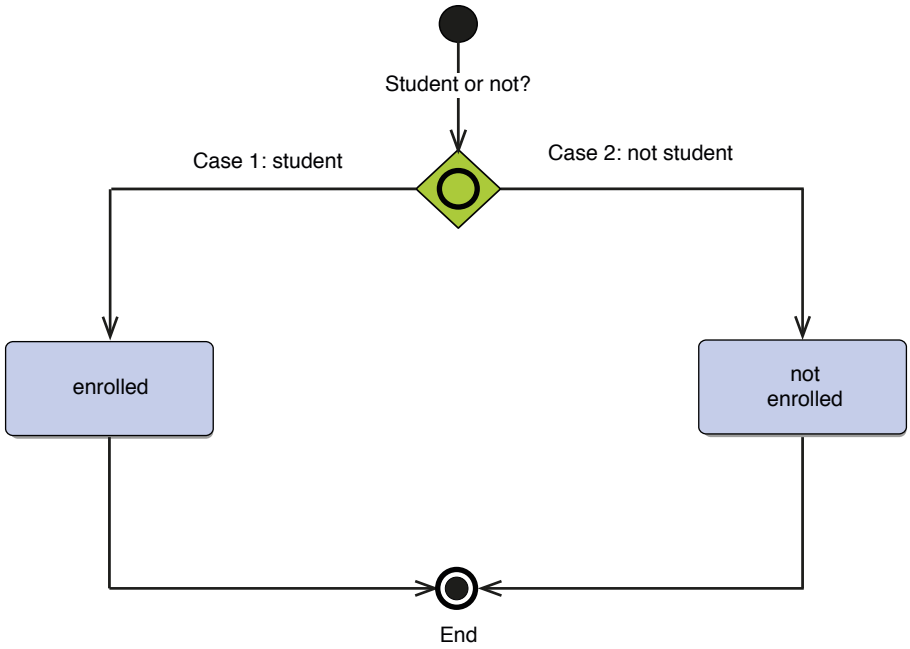


Figure 14.2: An example of a flow chart with a conditional statement.

The example has shown that a programming language must have at least one way of making such case distinctions in order to control the program flow. Java has adopted two constructs from C/C++ with the if and case statements. The if statement allows simple distinctions, while the case statement can differentiate between multiple states. In the following, you will learn how to use both constructs with the help of some examples.

14.3 »If Then Else« Statement

The if statement of the following example may look familiar and is in fact almost a duplicate of the program from chapter 13, »Operators«, section 13.5.4, »Ternary Operator«. The

purpose here is to check whether the value *student* is valid, that is, true. If it is, the variable *studentState* is assigned the value *is a student*, otherwise *is not a student*.

Listing 14.1: Two if-then-else constructs

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  class IfThenElseStatementDemo {
6
7    static String checkState(boolean student) {
8      if (student)
9        return " is a student";
10     else
11       return " is not a student";
12   }
13
14   public static void main(String[] arguments) {
15
16     // Person "Anna", the object identifier is anna:
17     Person anna = new Person("Anna", true);
18     String state = checkState(anna.isStudent());
19     System.out.println(anna.getName() + state);
20
21     // Person "Robert", the object identifier is robert:
22     Person robert = new Person("Robert", false);
23     state = checkState(robert.isStudent());
24     System.out.println(robert.getName() + state);
25   }
26 }
```

The statements are executed twice in this program with different results, because the two persons Anna and Robert each have a different status. The result of the program is:

```
Anna is a student
Robert is not a student
```

14.4 Ternary Operator

Here again the same example with the ternary operator. Here, too, we need to check whether the value *student* is valid, i.e. true. If this is the case, the variable *studentState* receives the value *is a student*, otherwise it receives the value *is not a student*.

Listing 14.2: A conditional statement by using a ternary operator.

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  class TernaryOperatorDemo {
6
7    static String checkState(boolean state) {
8      String stateDescription =
9        ((state) ? " is a student" : " is not a student");
10     return stateDescription;
11   }
12
13   public static void main(String[] arguments) {
14     // Person "Anna", the object identifier is anna:
15     Person anna = new Person("Anna", true);
16     String studentState = checkState(anna.isStudent());
17     System.out.println(anna.getName() + studentState);
18
19     // Person "Robert", the object identifier is robert:
20     Person robert = new Person("Robert", false);
21     studentState = checkState(robert.isStudent());
22     System.out.println(robert.getName() + studentState);
23   }
24 }
```

The statements are executed twice in this program with different results, because the two persons Anna and Robert each have a different status. The result of the program is again:

```
Anna is a student
Robert is not a student
```

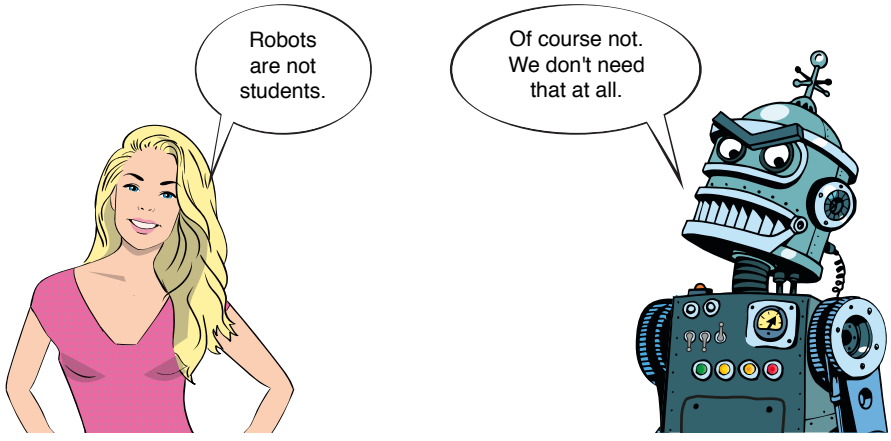


Figure 14.3: Robert believes that he does not need to study.

14.5 Switch Statement

14.5.1 Switch Statement at the Level of Java 6

If you have a lot of possibilities for a program branch, »If« branches as a solution quickly become confusing. As a substitute then the switch conditional offers itself. However, the variable following the keyword *switch* may only be of type *char, byte, short, int, enum* or *string* (since Java 7). For example, a truth value is not allowed.

Listing 14.3: The switch statement up to Java 6.

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  public class Java6SwitchDemo {
6
7    static String checkState(boolean student) {
8      int state = student ? 1 : 0;
9      String stateDescription;
10     switch (state) {
11     case 0:
12       stateDescription = " is not a student";
13       break;
14     case 1:
15       stateDescription = " is a student";
16       break;
17     default:
18       stateDescription = "";
19       break;
20     }
21     return stateDescription;
22   }
23
24   public static void main(String[] arguments) {
25     // Person "Anna", the object identifier is anna:
26     Person anna = new Person("Anna", true);
27     String studentState = checkState(anna.isStudent());
28     System.out.println(anna.getName() + studentState);
29
30     // Person "Robert", the object identifier is robert:
31     Person robert = new Person("Robert", false);
32     studentState = checkState(robert.isStudent());
33     System.out.println(robert.getName() + studentState);
34   }
35 }
```

If a case statement is to be exited when a condition is met, it *must* be terminated with a *break* must be used to terminate it. If this is forgotten, the program evaluates the next case statement. In this case, all breaks are present.

The program therefore outputs the following:

```
Anna is a student
Robert is not a student
```

14.5.2 Switch Statement at the Level of Java 7

As mentioned several times, Java 7 introduces the new feature that strings can also be used as arguments to the switch statement - something that has been allowed in other programming languages for years. You can see how strings are used in the context of the switch statement in the next example:

Listing 14.4: The Switch statement as of Java 7

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  public class Java7SwitchDemo {
6
7    static String checkState(String stateDescription) {
8      switch (stateDescription) {
9      case "student":
10       return " is a student";
11     case "no student":
12       return " is not a student";
13     default:
14       return "";
15     }
16   }
17
18   public static void main(String[] arguments) {
19     // Person "Anna", the object identifier is anna:
20     Person anna = new Person("Anna", "student");
21     String stateDescription = checkState(anna.getStateDescription());
22     System.out.println(anna.getName() + stateDescription);
23
24     // Person "Robert", the object identifier is robert:
25     Person robert = new Person("Robert", "no student");
26     stateDescription = checkState(robert.getStateDescription());
27     System.out.println(robert.getName() + stateDescription);
28   }
29 }
```

Also note the default statement. It is important so that you do not have to consider all cases that may occur. If you only want to handle one or two cases in the program, but want to make sure that all other cases are also considered, you must include this statement. It is generally recommended to safeguard the program against unexpected program states.

Of course, the sample program again outputs the following:

```
Anna is a student
Robert is not a student
```

14.5.3 Yield Statement

Since Java 12, there is also an elegant way to use the so-called yield statement in switch branches. It significantly shortens the processing in a switch branch (Listing 14.5).

Listing 14.5: The switch statement as of Java 12

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  public class YieldStatementDemo {
6
7    private static String determinePerson(int height){
8      return switch(height){
9        case 171: yield "Anna";
10       case 172: yield "Julia";
11       case 185: yield "Florian";
12       case 186: yield "Lukas";
13       case 190: yield "Robert";
14       default: yield "No idea";
15     };
16   }
17
18   public static void main(String[] args) {
19     System.out.println("Result:");
20     System.out.println("171 cm: " + determinePerson(171));
21     System.out.println("172 cm: " + determinePerson(172));
22     System.out.println("185 cm: " + determinePerson(185));
23     System.out.println("186 cm: " + determinePerson(186));
24     System.out.println("190 cm: " + determinePerson(190));
25     System.out.println("220 cm: " + determinePerson(195));
26   }
27 }
```

The sample program outputs the following:

```
Which height fits which person?
171 cm: Anna
172 cm: Julia
185 cm: Florian
186 cm: Lukas
190 cm: Robert
220 cm: No idea
```

14.5.4 Lambda Expression

Lambda expressions have been around since Java 8. Since Java 13, the lambda operator can be used in a switch branch, which again shortens the program. In listing **??** you can see the previous example in the new style.

Listing 14.6: The Switch construct from Java 13

```
1  //Examples/Conditional_Statements
2
3  package programmingcourse;
4
5  public class LambdaExpressionDemo {
6
7    static String determinePerson(int height) {
8      return switch(height) {
9        case 171 -> "Anna";
10       case 172 -> "Julia";
11       case 185 -> "Florian";
12       case 186 -> "Lukas";
13       case 190 -> "Robert";
14       default -> "No idea";
15     };
16   }
17
18   public static void main(String[] args) {
19     System.out.println("Result");
20     System.out.println("171 cm: " + determinePerson(171));
21     System.out.println("172 cm: " + determinePerson(172));
22     System.out.println("185 cm: " + determinePerson(185));
23     System.out.println("186 cm: " + determinePerson(186));
24     System.out.println("190 cm: " + determinePerson(190));
25     System.out.println("220 cm: " + determinePerson(220));
26   }
27 }
```

The sample program again outputs the following:

```
Which height fits which person?
171 cm: Anna
172 cm: Julia
185 cm: Florian
186 cm: Lukas
190 cm: Robert
220 cm: No idea
```

14.6 Summary

Conditional statements control the program flow. The decision in which direction the program branches is made on the basis of conditions. Java provides two different types of conditional statements for this purpose: if and case statements. For simple cases where the program needs to distinguish between two states, the if statement is perfectly sufficient.

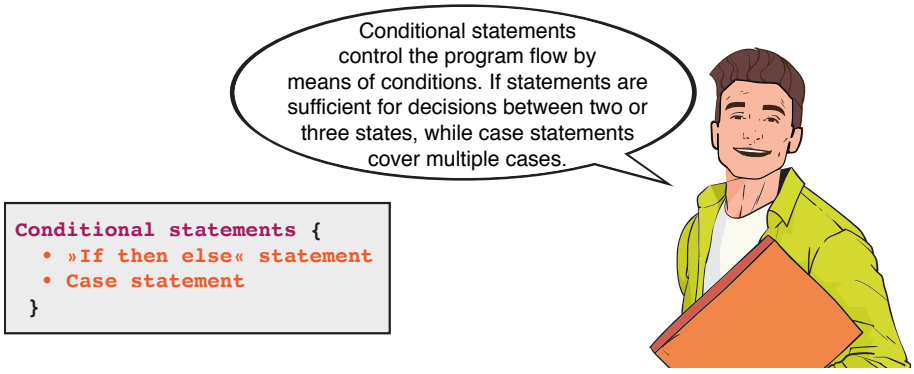


Figure 14.4: Conditional statements control the program flow by means of conditions.

In cases where the program needs to distinguish between many different states, it becomes confusing to combine many if statements. For this, the case statement is a better solution. Java 12 and 13 has brought some innovations for this statement that are very handy.

14.7 Literature

»Control Flow Statements«:
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html

14.8 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/conditional_statements

14.9 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/conditional_statements
There you will also find the solutions to the exercises.

15 Loops

15.1 Introduction

So far, this book has only introduced a lot of relatively »lifeless« data types and operators as parts of the Java programming language. Java programs become a little more dynamic when data is read in or output dynamically. This requires corresponding program structures – in short: everything that is generally understood in computer science as »loops«.

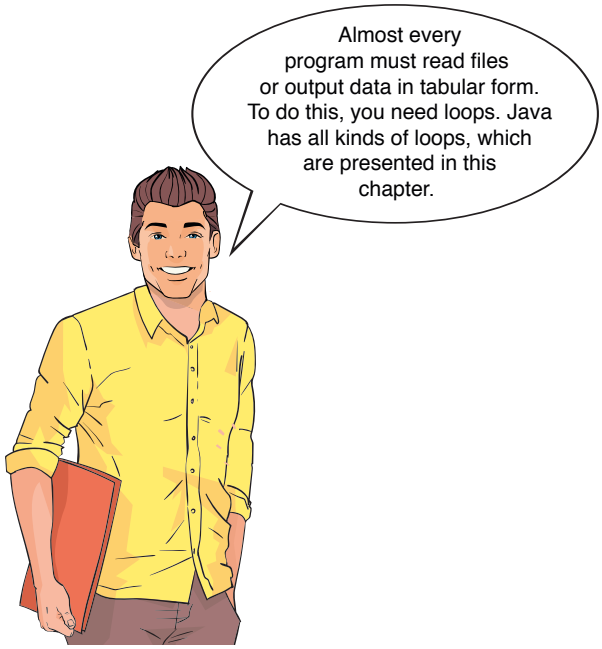


Figure 15.1: Loops are one of the most important statements of any programming language.

15.2 Overview

15.2.1 Purpose of Loops

Like conditionals, loops belong to the control structures of a programming language. They allow one or more statements to be repeated until a termination condition is met. Without a termination condition, the instructions would be trapped in an infinite loop. Loops thus combine the branches of the last chapter with the statements of chapter 7, »Statements«.

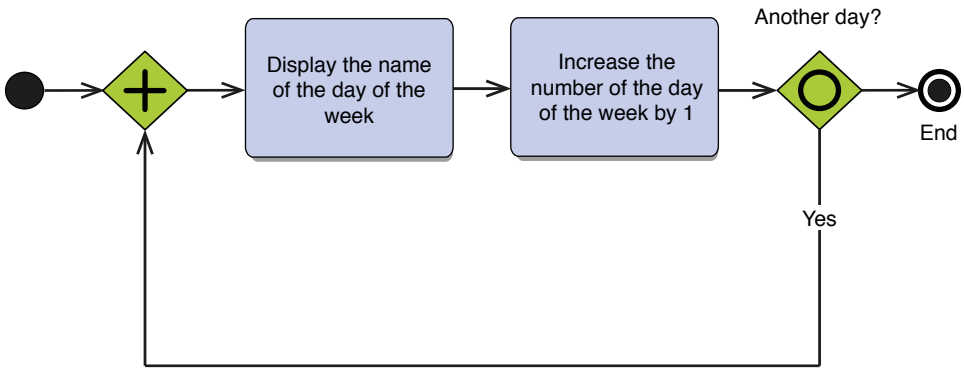


Figure 15.2: This loop outputs the name of Professor Roth's course participants.

You can see an example of a loop in figure 15.2. In this loop, the current number of the member of our programming course of Professor Roth is number one at the beginning. Then the program outputs the name of the current course member. In the next step, the program increments the current member number by one. Then, as a termination condition, it checks whether the total number of participants is greater than the current member number. If it is not, the program returns to the beginning and outputs the name of the next course member. Otherwise, the program ends.

15.2.2 Types of Loops

Java defines three different types of loops, with the for loop existing in two subtypes (Table 15.1).

Table 15.1: Types of Loops

Type of loop	Description
While loop	Number of runs previously unknown, rejecting
Do loop	Number of runs previously unknown, non-rejecting
Simple for loop	Number of runs known in advance, very fast output, rejecting
Extended for loop	Number of runs known in advance, very fast output, rejecting

15.3 While Loop

This type of loop belongs to the type of so-called head-controlled loops. The while loop is rejecting if the expression in the loop header is *false*. The loop will not run if the termination condition in the loop header is *false* (Figure 15.3).

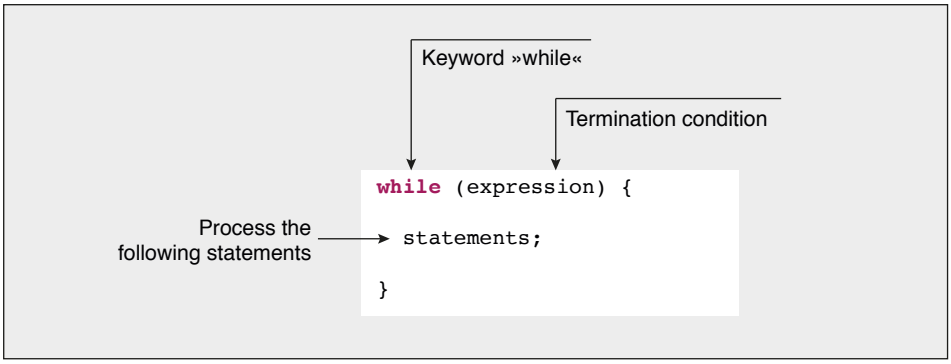


Figure 15.3: Structure of the while loop.

Listing 15.1 shows an example of a while loop. The following example outputs the days of the week one after the other, which were defined in a string array. Since the numbering of an array always starts at zero, the loop also starts there. In the head of the loop (line 16), the program asks whether *weekday* is smaller than the length of the array *weekdays*. If this is the case, the loop is executed the first time. Then the process repeats until the expression in the loop header is *true*, which means that the loop has reached the end of the array.

Listing 15.1: A While Loop.

```
1  //Examples/Loops
2
3  package programmingcourse;
4
5  class WhileLoopDemo {
6
7    static String[] weekdays = { "Monday", "Tuesday", "Wednesday",
8        "Thursday", "Friday", "Saturday", "Sunday" };
9
10   public static void main(String[] arguments) {
11     byte weekday = 0;
12     System.out.println("Days of the week:");
13     while (weekday < weekdays.length) {
14        System.out.println(weekdays[weekday]);
15        weekday++;
16     }
17   }
18 }
```

15.4 Do Loop

The do loop belongs to the »foot-controlled« loops. This type of loop is non-rejective. It checks whether the value of the expression is *true* or *false* before the first pass *not*. This means that the loop will be run even if the termination condition is met at the beginning. After the first run, the loop will of course be terminated correctly should the termination condition be met (Figure 15.4).

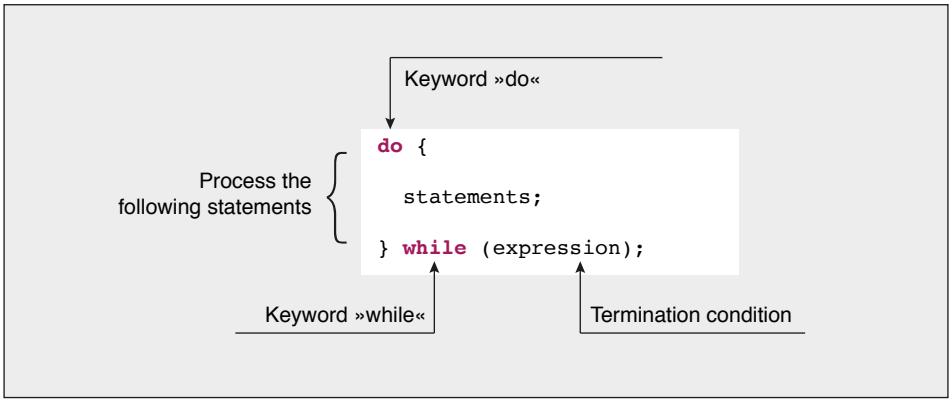


Figure 15.4: The structure of the do loop.

The program in Listing 15.2 shows an example of using the do loop. It again outputs in sequence the days of the week specified in a string array.

Listing 15.2: An example of a do loop.

```
1  //Examples/Loops
2
3  package programmingcourse;
4
5  class DoLoopDemo {
6
7    static String[] weekdays = { "Monday", "Tuesday", "Wednesday",
8        "Thursday", "Friday", "Saturday", "Sunday" };
9
10   public static void main(String[] arguments) {
11
12     byte weekday = 0;
13
14     System.out.println("Days of the week:");
15
16     do {
17       System.out.println(weekdays[weekday]);
18       weekday++;
19     } while (weekday < weekdays.length);
20   }
21 }
```

15.5 Simple For Loop

The for loop is also a rejecting type of loop. By »rejecting« is meant that in the head of the loop all conditions for execution are included. The first expression determines the initial value, the second the termination condition, and the third is a statement to control the termination condition.

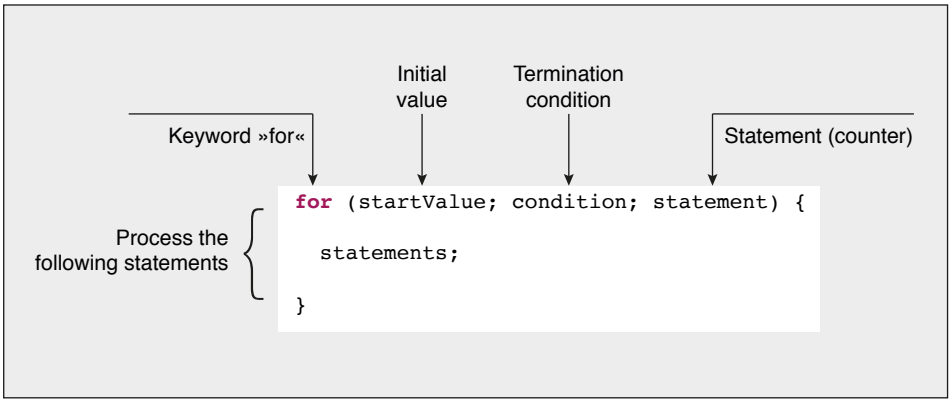


Figure 15.5: Structure of the simple for loop.

The following example again outputs in sequence the days of the week specified in a string array. Since the numbering of an array always starts at zero, the loop also starts with that. The loop ends when the end of the array has been reached. The loop determines this via the *day.length* method. As long as this value is not exceeded, the loop outputs the days of the week (Listing 15.3).

Listing 15.3: A simple For loop.

```
1  //Examples/Loops
2
3  package programmingcourse;
4
5  class SimpleForLoopDemo {
6
7    static String[] weekdays = { "Monday", "Tuesday", "Wednesday",
8        "Thursday", "Friday", "Saturday", "Sunday" };
9
10   public static void main(String[] arguments) {
11     System.out.println("Days of the week:");
12     for (int weekday = 0; weekday < weekdays.length; weekday++)
13       System.out.println(weekdays[weekday]);
14   }
15 }
```

15.6 Extended For Loop

Some Java developers found the programming of a for loop too cumbersome in certain cases. This is especially true when enums, lists or vectors are to be output. For this reason, the extended For loop has been available since Java 5. The structure of this loop can be seen in figure 15.6.

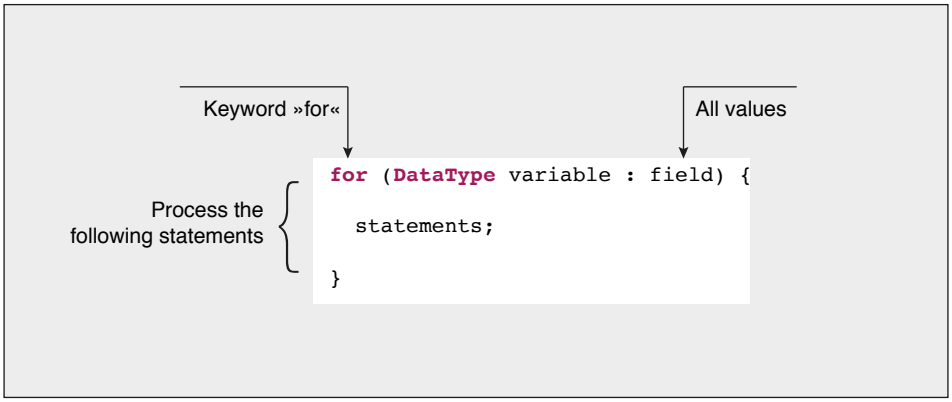


Figure 15.6: The structure of the extended For loop.

The extended form of the loop no longer needs an index. This loop type consists only of type, identifier and field. Translated, the loop is: »For all days of the week within the enum class *Weekdays*, output their names.«

Listing 15.4: An extended for loop.

```
1  //Examples/Loops
2
3  package programmingcourse;
4
5  public class ExtendedForLoopDemo {
6
7    private enum Weekdays {
8      Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
9    }
10
11   public static void main(String[] args) {
12     System.out.println("Days of the week:");
13     for (Weekdays weekday : Weekdays.values()) {
14       System.out.println(weekday);
15     }
16   }
17 }
```

The for loop, like all other loop types, can be interrupted with *break* and resumed with *continue*.

15.7 Summary

Loops allow recurring processes to be automated. Classic examples are reading in or outputting data. Java has three types of loops: the while loop, the for loop, and the do loop. The head-controlled while loop and the for loop are rejecting. That is, they check their termination criterion before running and thus reject a run if the termination criterion has already been reached.

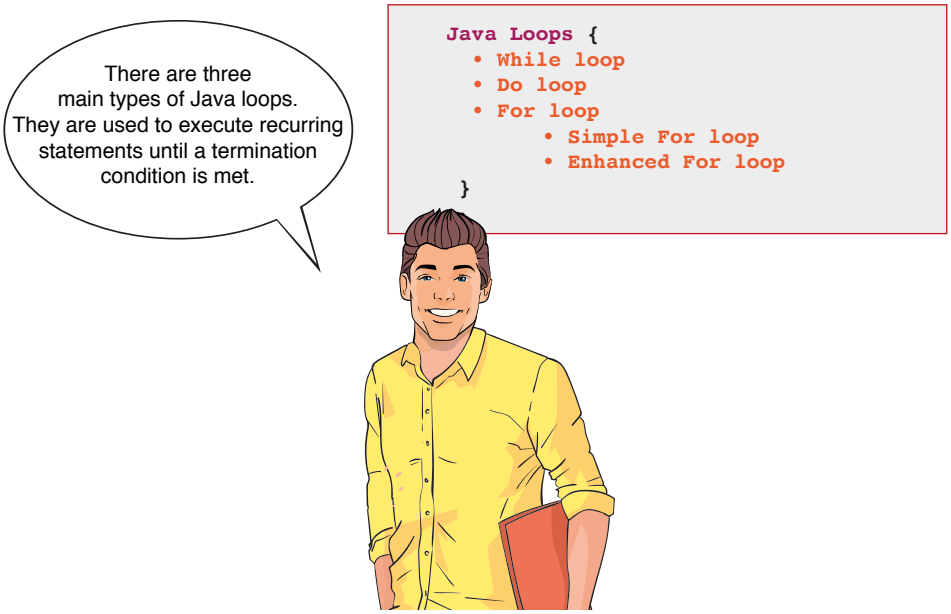


Figure 15.7: Java has three principally different types of loops.

The foot-controlled do loop, on the other hand, is not rejecting. The do loop therefore only checks after the first run whether the termination criterion has been reached. There are two versions of the for loop. Since Java 5 it is possible to use the extended for loop. This loop can, for example, automatically run through an enumeration type without an index.

15.8 Literature

Control Flow: *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html*

15.9 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/loops

15.10 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/loops
There you will also find the solutions to the exercises.

16 Packages and Modules

16.1 Introduction

In order to structure larger software systems, there are different concepts. In Java you solve the division of the programs over *Packages* and *Modules*. Packages bundle one or several classes, modules summarize one or several packages. While the package has been a component since the first version of the Java language, the module is very new in the Java area. It has been part of the language scope only since Java 9.

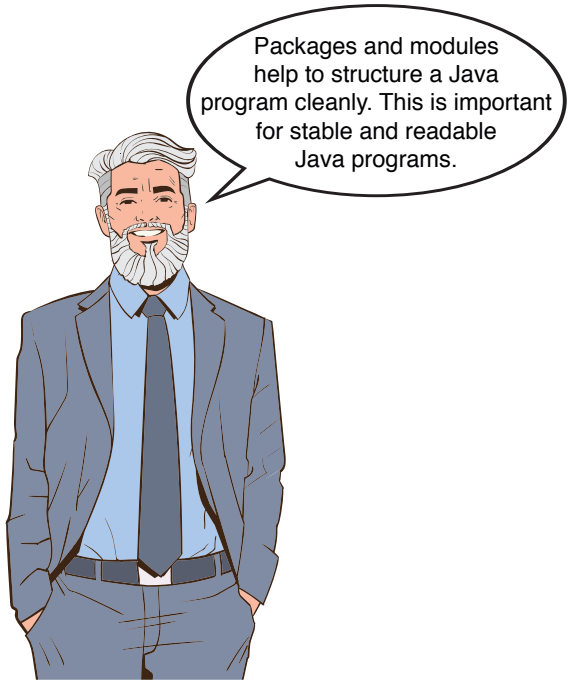


Figure 16.1: Professor Roth will help you better understand Java packages and modules.

16.2 Overview

Packages and modules are means to structure Java programs. Packages group one or more classes together, while modules group one or more packages together. Figure 16.2 shows the module *java.base* from Java SE, which bundles the packages *java.lang*, *java.math*, *java.net*, *java.io*, and *java.nio* and their subpackages.

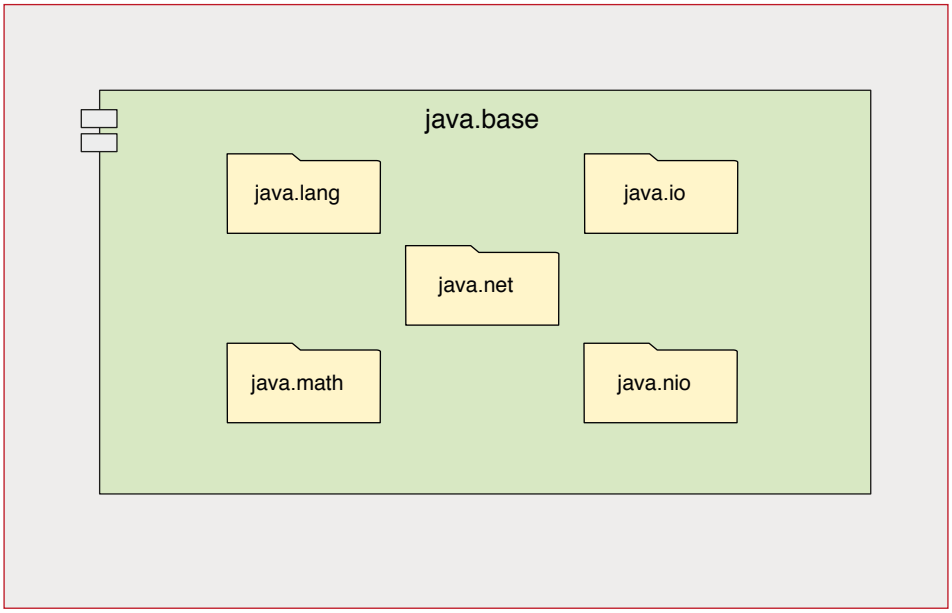


Figure 16.2: Java modules like »java.base« summarize one or more packages.

16.3 Packages

Packages are defined in Java using the keyword *package*. A single package contains one or more Java classes. The package should contain classes which are in *strong* relation to each other. To use classes outside of your own package, you have to import them.

16.3.1 Class Import

Dynamic Import

Packages are scopes for classes that reside within them. They are also called namespaces, which I will discuss later. Also public classes are unknown to other packages until they are taken over via an import statement. The term *import* is unfortunately misleading: The class is not imported, it is only *used*.

The import of classes can be executed either individually for each class of a package or for a whole package. In the latter case one uses a wildcard (Figure 16.3). There are some advantages to importing each class individually. This makes it easier for the programmer to understand which class was used.

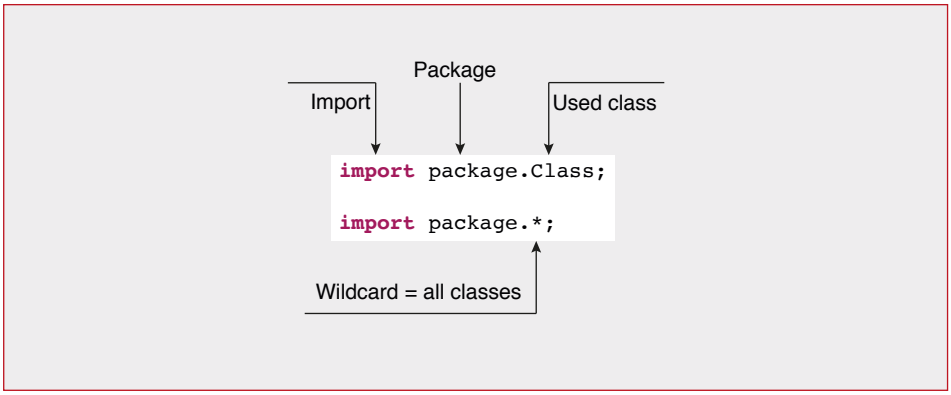


Figure 16.3: Structure of a dynamic import.

An example of this: Imagine the Roth family, consisting of the father Karsten Roth, who is separated from his ex-wife Katrin Stern. In Roth's »house«, the package *roth*, lives the daughter Katharina, in the other package *stern* the son Max from his first marriage. Both objects, *katharina* and *max*, are created from a class named *Child* (Figure 16.4).

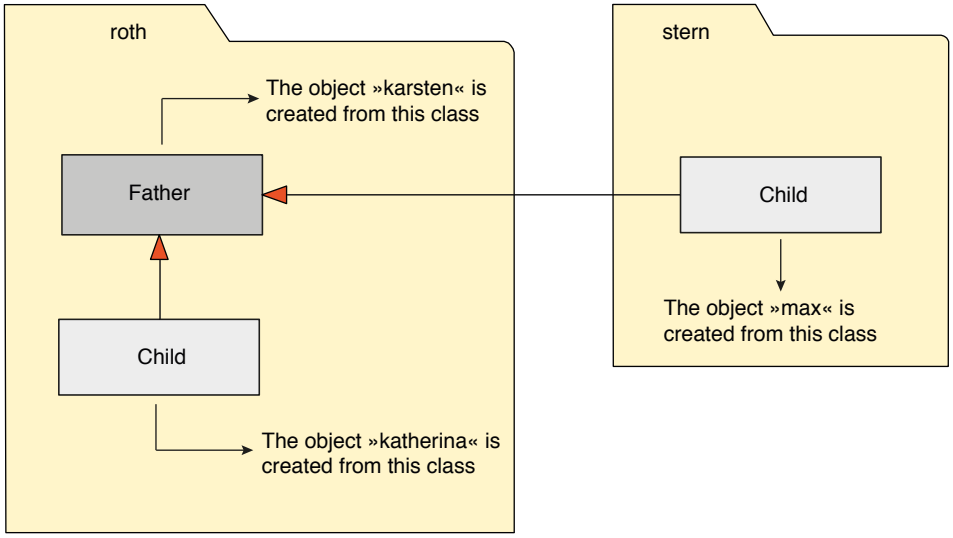


Figure 16.4: The class »child« has two meanings.

As you can see from Figure 16.4, the class *Child* strangely exists twice. What does this mean? In the left package *roth* it has the meaning of a child with strong relations to the father, while in the right package *stern* it has the status of a child with weak relations to the father.

In Listing 16.2, you can see an import statement in line 5. Through this statement, the class *Child* has access to the class *father*. To do this, the package must be specified with the full name.

Listing 16.1: The class »Child« is part of package »stern«.

```
1  //Examples/Packages_and_Modules
2
3  package stern;
4
5  import roth.Father;
6
7  public class Child extends Father {
8
9    public Child(String name) {
10     super(name);
11   }
12 }
```

In listing 16.1 things look different. The class *Child* automatically has access to the class *Father* without an import statement. This is because it is in the same package.

Listing 16.2: The class »Child« is part of package »roth«.

```
1  //Examples/Packages_and_Modules
2
3  package roth;
4
5  public class Child extends Father {
6
7    public Child(String name) {
8      super(name);
9    }
10 }
```

Java does not mind the duplicate definition of classes as long as each class of the same name is in a different package. This is why a package is commonly called a »namespace« in various programming languages. It restricts the access of a class to other classes (Chapter 23, »Rules«, section 23.4, »Access Protection«). How two classes with the same name are used in a main program is shown in Listing 16.3.

Listing 16.3: Both classes of the type »Child« are imported differently.

```
1  //Examples/Packages_and_Modules
2
3  package main;
4
5  import roth.Father;
6
7  public class PackageDemo {
8
```

```
9    public static void main(String[] args) {
10
11     Father karsten = new Father("Karsten");
12
13     stern.Child max = new stern.Child("Max");
14
15     roth.Child katharina = new roth.Child("Katharina");
16
17     System.out.println("Father: " + karsten.getName() + " (" +
18       karsten.getClass().getPackageName() + ")");
19
20     System.out.println("Daughter: " + katharina.getName() + " (" +
21       katharina.getClass().getPackageName() + ")");
22
23     System.out.println("Son: " + max.getName()  + " (" +
24       max.getClass().getPackageName() + ")");
25   }
26 }
```

The program shows in lines 13 and 15 what the two classes of type *Child* are called fully qualified. It can use the classes without an import statement because the program uses the fully qualified class name. The fully qualified class name consists of the complete package path including the class name.

Static Import

In conventional import statements, the class identifier must always be prefixed even when a static method is used (class method). For example, if you want to use mathematical functions such as the root operation, this interferes somewhat, as the following example in Listing 16.4 shows.

Listing 16.4: The conventional import of a class

```
1  //Examples/Packages_and_Modules
2
3  package programmingcourse;
4
5  import java.lang.Math;
6
7  public class DynamicImportDemo {
8
9    public static void main(String[] args) {
10
11    System.out.println("The square root is " + Math.sqrt(9));
12   }
13 }
```

The reason for this notation is clear: Java only allows the definition of methods that are bound to a class. Methods cannot exist detached from a class. And since there are no global methods, they must always be accessed in conjunction with the class. This is still

valid, except that the notation has been simplified by static imports since Java 5.0. Now methods can also be imported statically individually or via wildcards (Listing 16.5).

Listing 16.5: The static import of a class.

```
1  //Examples/Packages_and_Modules
2
3  package programmingcourse;
4
5  import static java.lang.Math.sqrt;
6
7  public class StaticImportDemo {
8
9    public static void main(String[] args) {
10
11    System.out.println("Die Quadratwurzel ist " + sqrt(9));
12   }
13 }
```

The notation leads to the fact that the method can be called without the associated class. The notation is very reminiscent of classic procedural programming languages (Figure 16.5).

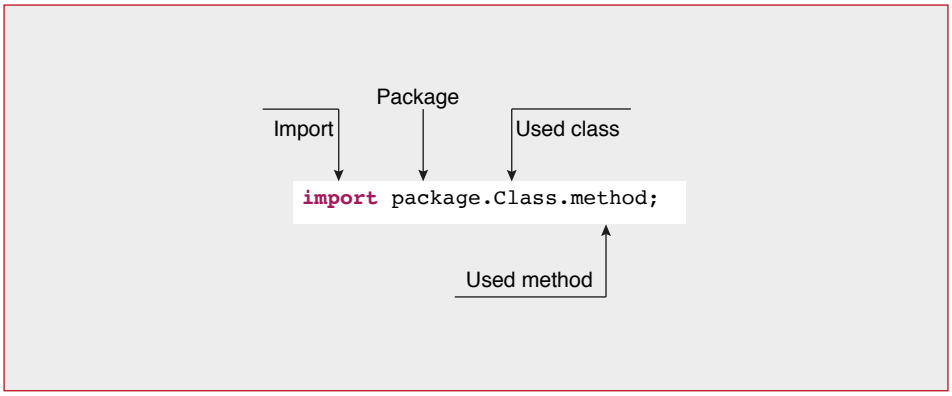


Figure 16.5: Structure of a static import.

16.4 Modules

Modules have only existed since Java 9. Oracle has postponed the release date several times because it was not so easy to break down the historically grown Java platform into individual modules. Java 9 introduced the term modules, one of the most important changes to the Java language. The official name of the new module system is Java Platform Module System (JPMS).

A dynamic module system allows different versions of a class to exist in a program. It also allows packages and classes from other projects to be used. A simple example: In a module named *novabot.robots*, there is a package named *robots*. The package *robots* contains a class *Robot*, which is to be made available to other projects. For this purpose, the module contains a file named *module-info.java*, which contains the statement *exports robots*.

This statement ensures that the entire package of the module is exported. Another class can now specify that it requires to use the module *novabot.robots* with another text file called *module-info.java*. This allows it to access the exported packages of the module *novabot.robots*, but only these. The interaction can be seen in Figure 16.6.

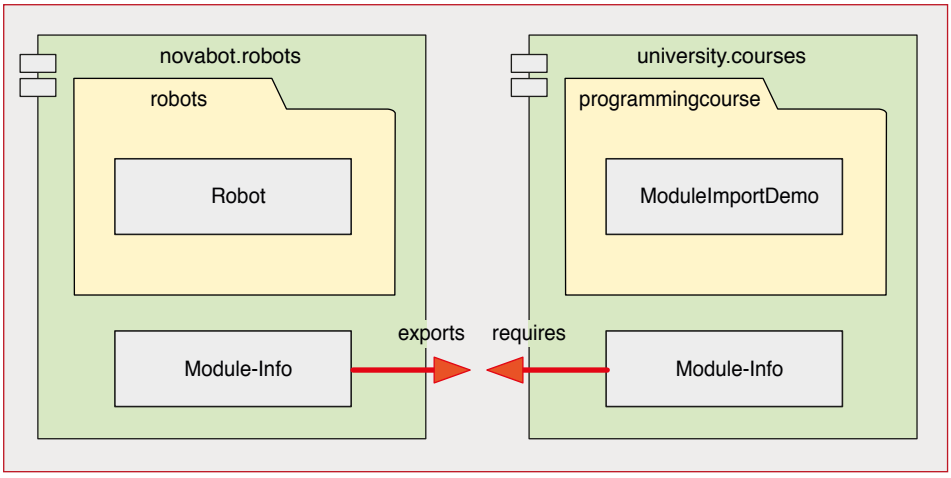


Figure 16.6: The module »novabot.robots« is exported.

The class *Robot* has a simple structure and consists only of a constructor and a getter method (Listing 16.6). You can find this class in the project *ModuleExportDemo*.

Listing 16.6: The class »Robot« belongs to the module »novobot«.

```
1  //Examples/Packages_and_Modules
2
3  package robots;
4
5  public class Robot {
6
7    private String name;
8
9    public Robot(String name) {
10     this.name = name;
11   }
12
13   public String getName() {
14     return name;
15   }
16 }
```

The class is protected by the module *novabot.robots*. To publish the class, we need a file named *module-info.java* with an export statement (Listing 16.7). The export removes the access protection of the module and exports the package *robots* from the module.

Listing 16.7: The file »module-info.java« from the module »novatec.robots«.

```
1  //Examples/Packages_and_Modules
2
3  module novabot.robots { // Module novatec.robots
4    exports robots; // Exported package
5  }
```

To use the exported package *robots*, another project also needs a file named *module-info.java* (Listing 16.8). This requires the module *novatec.robots* and thus access to the class *Robot*.

Listing 16.8: The file »module-info.java« from the module »university.courses«.

```
1  //Examples/Packages_and_Modules
2
3  module university.courses { // Module university.courses
4    requires novabot.robots; // Module novabot.robots is required
5  }
```

Finally, you can see the use of the class *Robot* in a main program (Listing 16.9). The program calls the class *Robot* from the other project as if it were part of its own project.

Listing 16.9: The class »ModuleImportDemo« imports a package.

```
1  //Examples/Packages_and_Modules
2
3  package programmingcourse;
4
5  import robots.Robot;
6
7  public class ModuleImportDemo {
8
9    public static void main(String[] args) throws ClassNotFoundException {
10     Robot robert = new Robot("Robert");
11     System.out.println("The robot '" + robert.getName() + "'" + " is " +
12       "delivered by '" + robert.getClass().getModule().getName() + "'");
13   }
14 }
```

The program outputs the following:

```
The robot 'Robert' is delivered by 'novabot.robots'
```

16.5 Summary

Packages and modules can be used to structure Java programs into manageable units. Packages contain one or more related classes, while modules bundle one or more packages.

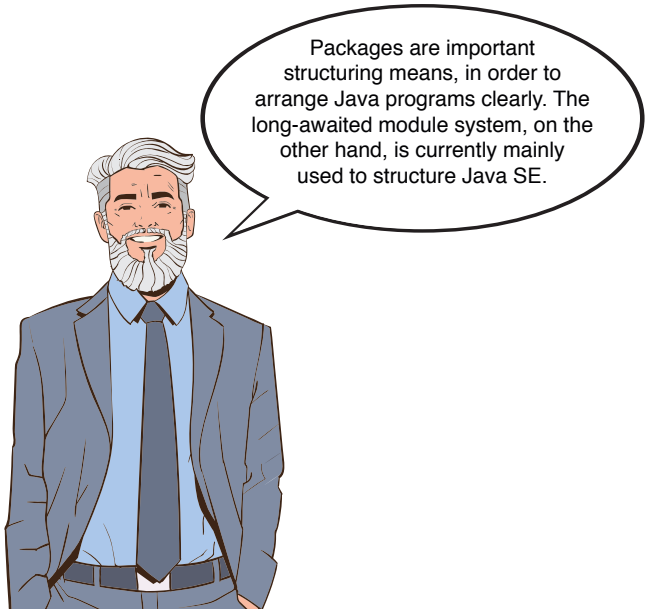


Figure 16.7: Packages are indispensable for structuring Java programs.

16.6 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/packages_and_modules

16.7 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/packages_and_modules
There you will also find the solutions to the exercises.

17 Exception Handling

17.1 Introduction

Nobody likes errors – especially not in computer programs. But what happens when they occur anyway? How can these errors be caught and handled? In classical programming languages one helps oneself with error numbers. In contrast, error handling in Java is purely object-oriented.

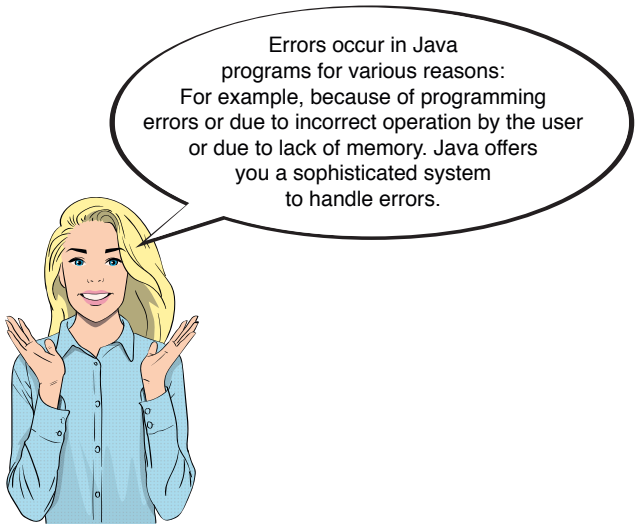


Figure 17.1: Who likes errors? But when they occur, they should be handled confidently.

The so-called »Exception Handling« is the cause of many misunderstandings. Therefore, this chapter introduces you to handling of exceptions in Java programs in great detail as one of the most important features of the programming language. A good understanding of exception handling is the key to very robust Java programs.

17.2 Overview

17.2.1 Motivation

Exception handling is a principle embedded in Java to respond to exception conditions in a program. This is necessary to catch and react to errors within a program cleanly. Such exception states of an application can be divided into two categories: business and technical errors.

17.2.2 Types of Errors

Domain-oriented errors occur, for example, when the user performs a domain-oriented action that is incorrect. An example of a technical error is entering a letter in a number field or entering a number that is too large. The incorrect entries should not lead to an instability of the program, but should be caught by the program confidently via technical checks. The program should also react to incorrect entries with meaningful hints. Figure 17.2 shows how a Java computational program reacts to the input of a letter.

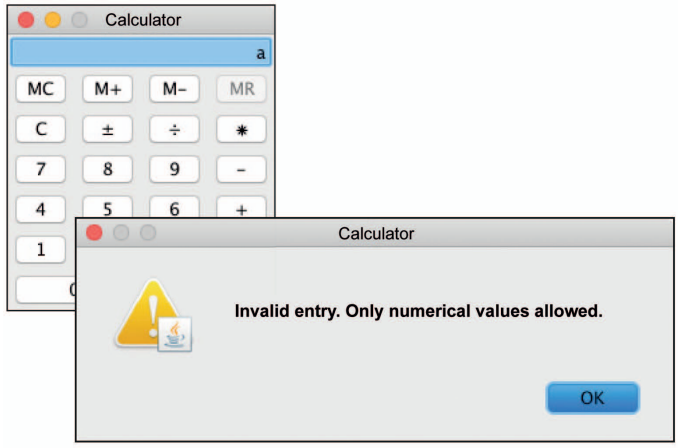


Figure 17.2: A program should respond to incorrect input with meaningful hints.

Technical errors are given, for example, when there is too little memory available, a class was not found by the runtime environment, or a read/write error occurred. Here is another example: Suppose you want to write a program that calls another program. If you were writing a Java program, you would call an appropriate method. This method might fail if the file to be executed has been moved and is therefore not found by the program or is destroyed.

17.2.3 Use of Exception Handling

Error handling in Java runs in an object-oriented manner (Figure 17.3). The starting point is always that the Java program *attempts* to execute a method (item 1). Due to the fact

that only an attempt is made here, of which it is unclear whether it succeeds, this call is surrounded by a try block. If the action goes well, the block is exited and the program simply continues to run.

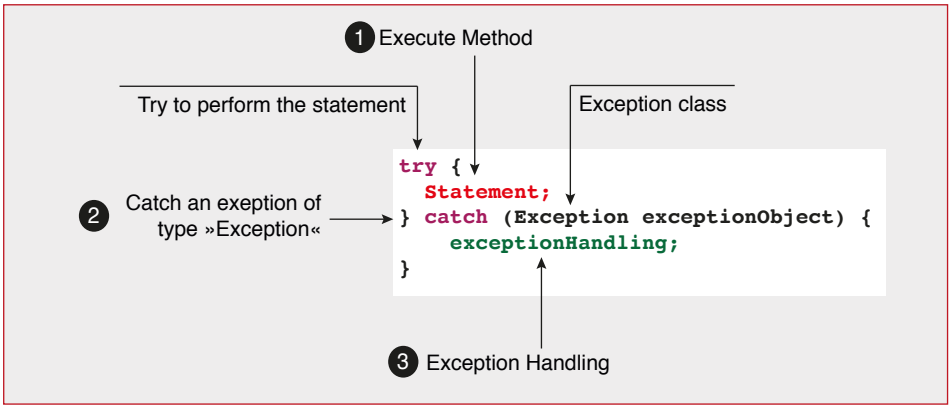


Figure 17.3: Principle process flow of error handling.

However, if this action fails, the method sends an object. This object belongs to the class *Exception* or a derived class. The program »catches« this object in a catch block with error handling (point 2). Since the error should be an »exception«, it is called »exception handling«. All classes involved in this are called »exceptions«.

A Try block can surround several statements, and there can also be several Catch blocks in a row. There is also a Finally statement. It is used to include statements that must be processed in any case. Important cleanup tasks can be placed here to avoid damage.

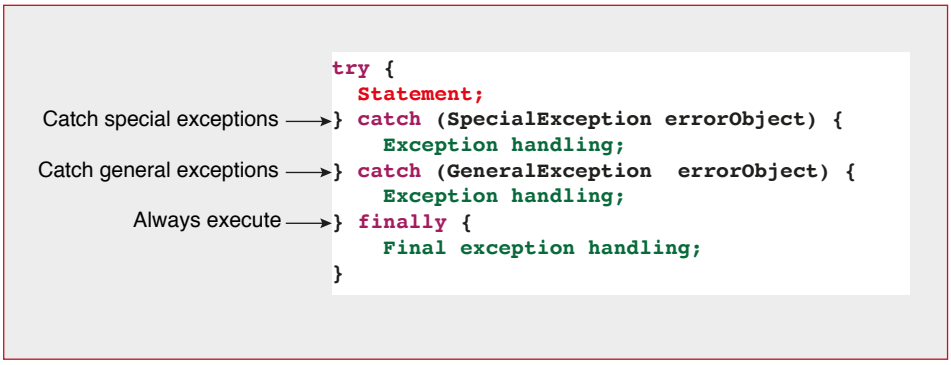


Figure 17.4: The extended error handling with a finally block.

If there can be multiple exception states in the try block, you must also catch multiple exception objects. The *correct order* of the catch blocks is important here. Catch the special exceptions first, and put the general exception (if needed) at the end (Figure 17.4). The following example shows why the order of processing is so important.

Listing 17.1: The order of catch blocks is very important,

```
try {
  // Step 1: open file - may throw FileNotFoundException;
  // Step 2: parse file - may throw general exception;
} catch (FileNotFoundException fileNotFound) {
  // show dialog with message that
  // a file was not found;
} catch (GeneralException generalException) {
  // General error handling instructions;
} finally {
  // instructions that must be completed
}
```

In *step 1*, the program is supposed to read a file. This action may cause an error to occur if the file does not exist. In this case, the program should alert the user by displaying a dialog. On the other hand, if the action went well, *step 2* is executed as scheduled. If the order of the catch blocks is reversed, the listing looks like this:

Listing 17.2: Execute with catch blocks swapped.

```
try {
  // Step 1: open file - may throw FileNotFoundException;
  // Step 2: parse file - may throw general exception;
} catch (GeneralException generalException) {
  // General error handling instructions;
} catch (FileNotFoundException fileNotFound) {
  // Indicate error that a file was not found;
} finally {
  // Instructions that must be completed
}
```

If the program executes *statement 1* and fails to find a file, the first catch block takes effect. The exception condition is already caught by the first block because the error object of type *FileNotFoundException* is also an object of type *Exception*. The *Exception* class is the base class for all exceptions, including *FileNotFoundException*. What is the consequence? Instead of informing the user correctly, the program can only output that some error happened.

How are error objects created? The creation of exceptions is defined in a class. The developer of this class has foreseen that a method of his class can cause an error. Figure 17.5 shows an example of a method. The method header contains the keyword *throws* to declare an error object (item 1). This is information for the developer who wants to use this function. It indicates which exception(s) the program must catch when the method is called.

In point 2 of the figure, you can see how the class checks if an error has occurred. If it has, it creates a new error object using the *throw* statement. Here it calls the constructor of an error class as usual with the new operator.

Exception handling is designed to protect programs against errors. Through the procedure, a development environment is able to force the programmer to handle potential exceptions in a program. Figure 17.6 shows this. Here, the program attempts to call the *checkstate(int state)* method. This method may throw an exception, which the program does

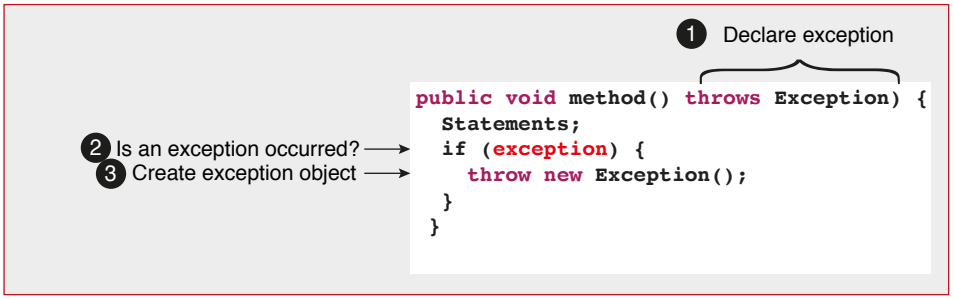


Figure 17.5: Any java class can send error objects if needed.

not handle in this example. Unless the programmer changes this and includes a sufficient try-catch statement, Eclipse does not compile the program.

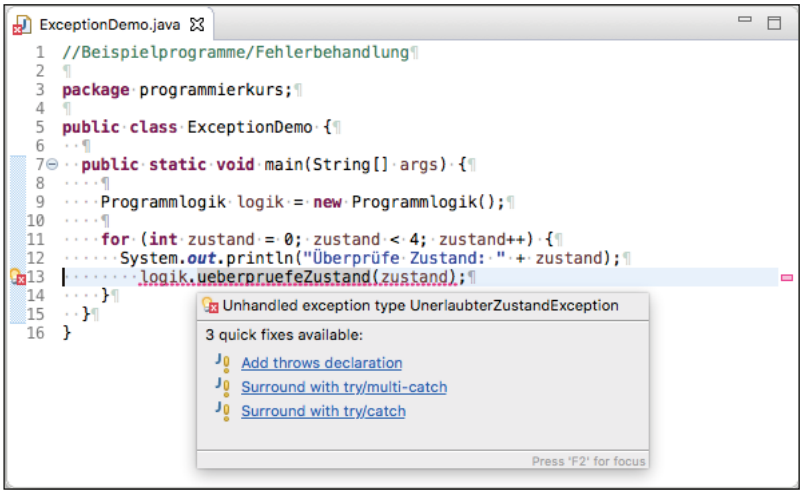


Figure 17.6: Eclipse warns about unhandled exceptions.

17.3 Base Class »Throwable«

Within the Java Standard Edition, there is a widely branched tree of exception classes that are used by other Java classes to create error objects. The classes originate in the class *Throwable*, which, like all Java classes, descends from the superclass called *Object*. From *Throwable* derive the base classes *Exception* and *Error* (Figure 17.7).

Exceptions belonging to the branch *Error* of the class tree are thrown when a critical condition of the Java Runtime Environment has occurred. There is little point in trying to catch these errors during program runtime. Instead, you should rule out such errors when designing the program. In contrast, the *Exception* subtree are classes whose errors you can

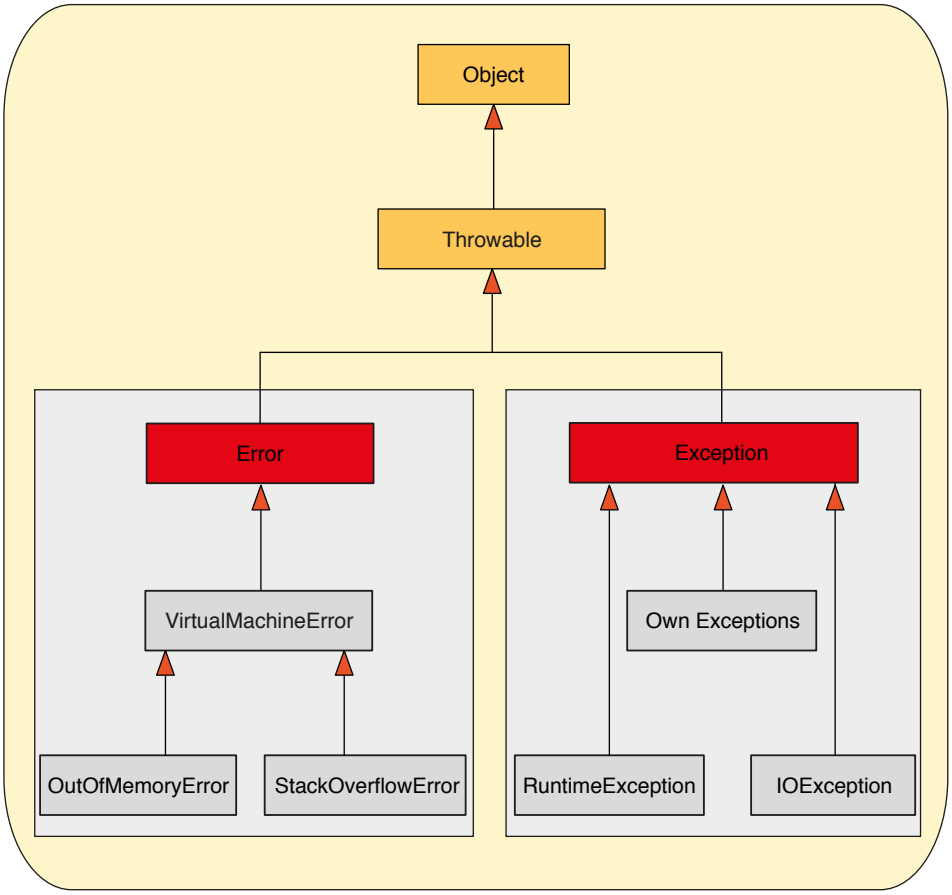


Figure 17.7: All errors can be assigned to the derived classes »Error« and »Exception«.

catch and handle at runtime. Let's dive a little further into the class tree and start with errors associated with the *Error* class and its underlying classes.

17.4 Class »Error«

Errors of type *Error* indicate a serious error within a Java program or the configuration of the Java program, respectively. As mentioned before, it is not very useful to try to fix these errors during the runtime of a program. Instead, you must ensure that such errors do not occur under any circumstances before a program is delivered. Figure 17.8 shows a snippet of the most important error classes below the base class *Error*.

The two classes derived from *VirtualMachineError* *OutOfMemoryError* and *StackOverflowError* can occur in Java programs in rare cases. This happens when the Java program and the Java Virtual Machine (JVM) lack memory. The reason for this lack of memory can be,

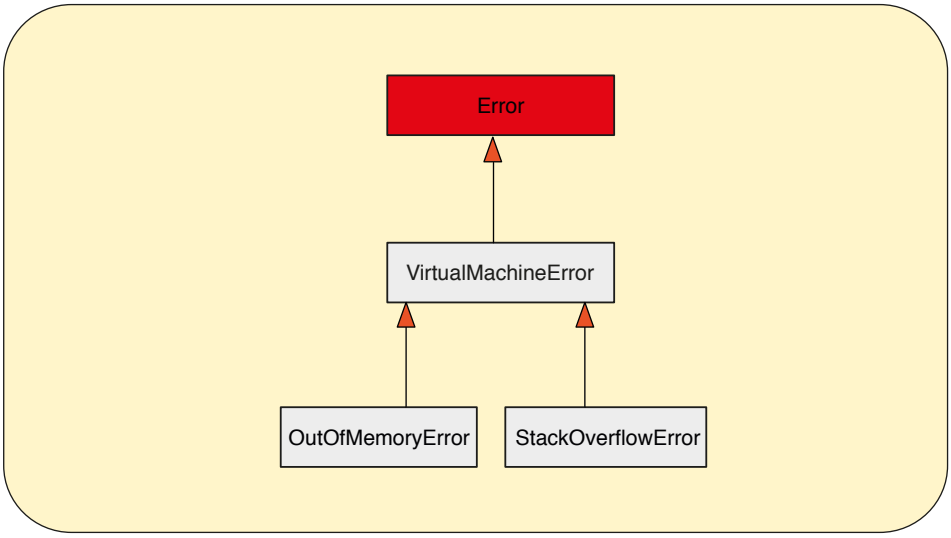


Figure 17.8: The base class »Error« and a snippet of the main derived subclasses.

for example, that there is a »programming« or configuration error. To better differentiate the reasons for this lack of memory, the Java Virtual Machine distinguishes between *OutOfMemoryError* and *StackOverflowError*.

17.4.1 Subclass »OutOfMemoryError«

Errors of type *OutOfMemoryError* have the very technical reasons listed in table 17.1. These »programming« or configuration errors should be avoided before program delivery. For example, a programming error occurs when the program is designed to use more and more memory without the JVM's garbage collector being able to free it (»Requested Array Size Exceeds VM Limit«).

A configuration error may well be present in larger Java programs if the program already reports a lack of memory during normal operation (»Java Heap Space«). If, however, the user can open very many windows or load graphics, for example, one must consider whether one wants to limit this or generally allow the program more memory. In the latter case, one must allocate more memory to the JVM. Chapter 21, »Runtime Environment«, section 21.8, »JVM Configuration«, goes into detail about these settings.

To give you a better idea of how such errors can occur in Java programs, a few examples. Let me start with a program that provokes an error of type *OutOfMemoryError* due to insufficient Java heap space (17.4).

Listing 17.3: This program triggers an OutOfMemoryError.

```
1 //Examples/Exception_Handling
2
3 package programmingcourse;
```

Table 17.1: Types of OutOfMemory errors.

Error message	Reason	Solution
Java Heap Space	There is not enough memory available in the heap	increase heap size (parameter at JVM startup)
GC Overhead Limit Exceeded	The garbage collector is executed too often	increase heap size (parameter at JVM startup)
Requested Array Size Exceeds VM Limit	The VM limit for the size of an array is exceeded	Either increase heap size (configuration) or check program allocation (program error)
Metaspace	There is not enough memory left for class metadata	decrease heap size if possible. This will increase the memory for metadata
Request Size Bytes For Reason. Out of swap space?	Mapping to native heap failed	Here you have to check if the computer provides enough main memory in general
Compressed Class Space	Memory for compressed classes exhausted	Increase CompressedClassSpaceSize gradually
Reason Stack_Trace_ With_Native_Method	An allocation error occurred for a native method	In case of such an error, you should consult Oracle guidance (see section 17.7, »Literature«).

```
4
5  public class OutOfMemoryErrorHeapSpaceDemo {
6
7    public static void main(String[] args) {
8      int[] array = new int[1000000000];
9      System.out.println(array[0]);
10   }
11 }
```

When you run the program, you will get the following message after a short time:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at programmingcourse.OutOfMemoryErrorHeapSpaceDemo.main(
            OutOfMemoryErrorHeapSpaceDemo.java:6)
```

The message means that the Java Virtual Machine (JVM) has thrown an out-of-memory error in line 6 of the program due to lack of »Java Heap Space«. The cause is that the program is completely careless in trying to create a huge array. With this program construction, the JVM's garbage collector has no chance whatsoever to release memory that is no longer needed. In other words, this is a bad programming mistake - just as in the next example, which carelessly exhausts the JVM's memory limit.

Listing 17.4: This program throws an »OutOfMemoryError«.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
```

```
4
5  public class OutOfMemoryErrorVMLimitDemo {
6
7    public static void main(String[] args) {
8      int[] array = new int[Integer.MAX_VALUE];
9      System.out.println(array[0]);
10   }
11 }
```

When you run the program, you immediately get the following error:

```
Exception in thread "main" java.lang.OutOfMemoryError: Requested array size
    exceeds VM limit
        at programmingcourse.OutOfMemoryErrorVMLimitDemo.main(
            OutOfMemoryErrorVMLimitDemo.java:6)
```

What happened? This time the program tried to create an even bigger field than in the example before. The field this time is even larger than the JVM would allow, even if you allocate an extremely large amount of memory to the program. Unfortunately, the Eclipse development environment did not catch this error right away, resulting in an annoying runtime error.

17.4.2 Subclass »StackOverflowError«

Errors of type *StackOverflowError* occur when the virtual machine stack goes off the rails. With each method call, a Java program reserves data on a so-called call stack. The data includes, for example, the method's parameters. If a method calls itself again (recursion), too many calls may cause the stack's memory to be exceeded and the program to throw an error of type *StackOverflowError*, as shown in the example in Listing 17.5.

Listing 17.5: This program triggers an OutOfMemoryError.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  public class StackOverflowErrorDemo {
6
7    public static void kehreZurueck(int nummer) {
8      System.out.println("Rückkehr: " + nummer);
9
10     if (nummer == 0)
11       return;
12     else
13       kehreZurueck(++nummer);
14   }
15
16   public static void main(String[] args) {
17     StackOverflowErrorDemo.kehreZurueck(1);
18   }
19 }
```

17.5 Class »Exception«

Exceptions literally means exceptions. They are events that occur in case of errors. They can be runtime errors or other errors within a program. Figure 17.9 again shows you a section of the class hierarchy with the most important errors that belong to the base class *Exception*.

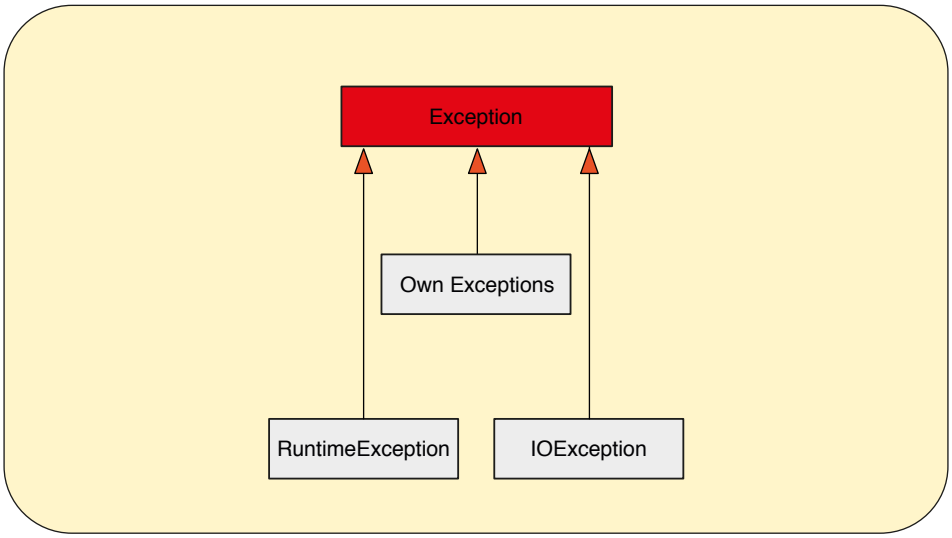


Figure 17.9: The base class »Exception« and a snippet of the main derived sub classes.

17.5.1 Subclass »RuntimeException«

Errors of type *RuntimeException* occur when programming errors have gone unnoticed. At program runtime, hence the name of the *RuntimeException* class, these errors then come to light. You cannot more usefully query these errors at runtime. Instead, you should make sure that your program does not generate such errors in the first place.

17.5.2 Subclass »IOException«

Unlike errors of type *RuntimeException*, errors of type *IOException* are almost unavoidable in many Java programs. They occur, for example, when a device fails, a drive is not available, or a file does not exist although it should. Therefore, these errors can also be queried during runtime. How this works is shown by an anticipation of the class *Runtime*. It is first introduced in chapter 22, »Class Libraries«. Among other things, it allows you to run an external program from a Java program. For example, on Windows, if you want to start the *Notepad.exe* program, you can implement it as shown in listing 17.6.

Listing 17.6: Process of error handling in a Java program.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  import java.io.IOException;
6
7  class IOExceptionDemo {
8
9    @SuppressWarnings("deprecation")
10   public static void main(String[] arguments) {
11
12     // Try to a start notepad.de (Windows only)
13     try {
14       Runtime.getRuntime().exec("notepad.exe");
15     }
16     catch (IOException exception) {
17       System.out.println(exception);
18     }
19   }
20 }
```

The sample program consists of a method *main()* in which the method *exec* is called with the parameter *'notepad.exe'*. The call to the function is in a try-catch block. This block stakes out the »problem zone« of the program: the method *exec()* may fail because the *Notepad.exe* program may not be found. Therefore, the call must be secured because of the risk.

17.5.3 Self-Programmed Exceptions

You can not only handle exceptions in your programs that are thrown by foreign classes. Instead, you can also define your own exceptions. The following program *ExceptionDemo* shows how an exception is thrown each time the state *state* moves outside a valid scope. If the state is 0 or 1, everything is fine. If the state is 2, an *CriticalStateException* is thrown. For all other values, the class throws a *UnknownStateException*.

Listing 17.7: This class can throw two exceptions.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  public class ProgramLogic {
6
7    public void checkState(int zustand)
8        throws UnknownStateException,
9        CriticalStateException {
10     switch (zustand) {
11       case 0:
```

```
12          System.out.println("The state is not critical.");
13          break;
14        case 1:
15          System.out.println("The state is critical.");
16          break;
17        case 2:
18          throw new CriticalStateException();
19        default:
20          throw new UnknownStateException();
21      }
22    }
23  }
```

The class uses two simple exceptions. The first of our two classes is called *CriticalStateException*.

Listing 17.8: This class defines an exception for an unallowed state.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  public class CriticalStateException extends Exception {
6
7    private static final long serialVersionUID = 1L;
8
9    public CriticalStateException(String message) {
10     super(message);
11   }
12
13   public CriticalStateException() {
14     super("The state is critical.");
15   }
16 }
```

The second class is needed when a more unknown condition has occurred.

Listing 17.9: This class defines an exception for an unknown state.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  public class UnknownStateException extends Exception {
6
7    private static final long serialVersionUID = 1L;
8
9    public UnknownStateException() {
10     super("The state is unknown.");
11   }
12 }
```

The three classes are used in a simple program that includes the try-catch block mentioned earlier:

Listing 17.10: This class demonstrates how exceptions are caught.

```
1  //Examples/Exception_Handling
2
3  package programmingcourse;
4
5  public class ExceptionDemo01 {
6
7    public static void main(String[] args) {
8
9      ProgramLogic logic = new ProgramLogic();
10
11     try {
12       for (int state = 0; state < 4; state++) {
13         System.out.print("Check state " + state + ": ");
14         logic.checkState(state);
15       }
16     } catch (CriticalStateException exception) {
17       System.out.println(exception.getMessage());
18     } catch (UnknownStateException exception) {
19       System.out.println(exception.getMessage());
20     }
21   }
22 }
```

The program modifies the value of *state* in a loop and passes it to the *checkstate()* method. If this results in an out-of-range condition, the method throws an exception, as shown by the program output for the value 2:

```
Check state 0: State is not critical.
Check state 1: state is not critical.
Check state 2: state is critical.
```

An exception for state 3 does not occur because the loop is exited after the first exception. This can be prevented by a so-called multi-catch block. Here is an example as well:

Listing 17.11: An example of a multi-catch block.

```
1  //Examples/Error_Handling
2
3  package programmingcourse;
4
5  public class ExceptionDemo02 {
6
7    public static void main(String[] args) {
8
9      ProgramLogic logic = new ProgramLogic();
10
11       for (int state = 0; state < 4; state++) {
```

```
12  //        System.out.print("Check state " + state + ": ");
13          try {
14            logic.checkState(state);
15          } catch (UnknownStateException
16              | CriticalStateException exception) {
17            System.out.println(exception.getMessage());
18          }
19        }
20    }
21  }
```

The program again loops to change the value of *state* and passes it to the *checkstate()* method. If a range overflow occurs, this program also catches the thrown exception. However, due to the multi-catch, this time the loop is not aborted by the first exception, so all error states can be seen:

```
Check state 0: state is not critical.
Check state 1: state is not critical.
Check state 2: state is critical.
Check state 3: state is unknown.
```

17.6 Summary

Exception handling is one of the most important aspects of programming. In Java, the handling of errors is divided into two parts. Via a try block one tries to call methods, which in case of an error lead to an exception, which one handles in the catch block. To be able to distinguish correctly between the error types, it is important to process errors from the fine to the coarse, from the specific to the unspecific.

Java already has a lot of ready-made exception classes that can be used in your own programs. Where this is not sufficient, you develop your own exception classes for error conditions that are program-specific. You will learn how this works with Eclipse in the following exercise.

17.7 Literature

Trouble Shooting:
https://docs.oracle.com/en/java/javase/13/troubleshoot/general-java-troubleshooting.html

```
try {
  Statement;
  } catch (SpecialException exceptionObjekt) {
    Special Exception Handling;
  } catch (GeneralException exceptionObjekt)
    General Exception Handling;
  }
}
```

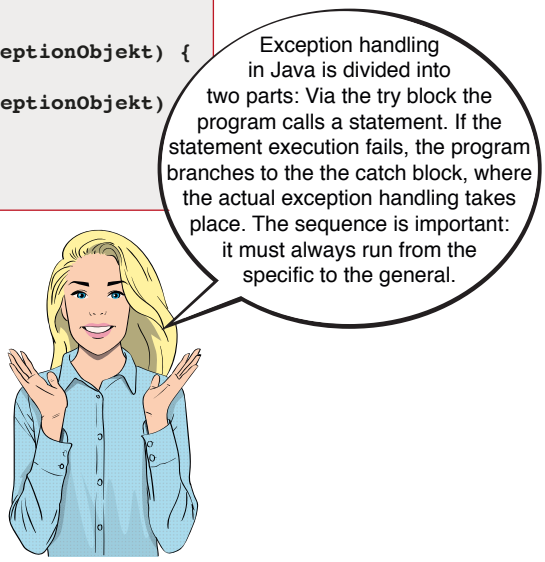


Figure 17.10: Anna hates programs that sweep bugs under the rug.

17.8 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/exception_handling

17.9 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/exception_handling
There you will also find the solutions to the exercises.

18 Documentation

18.1 Introduction

Although the topic may seem boring to you, programs need to be documented. This is not only to make sure that *others* understand your programs. It also serves to make sure that you *yourself* still understand your program after a certain time. This may seem paradoxical, but especially for larger projects with a lot of time pressure it is important that you also document for yourself to be able to maintain the program more easily later.

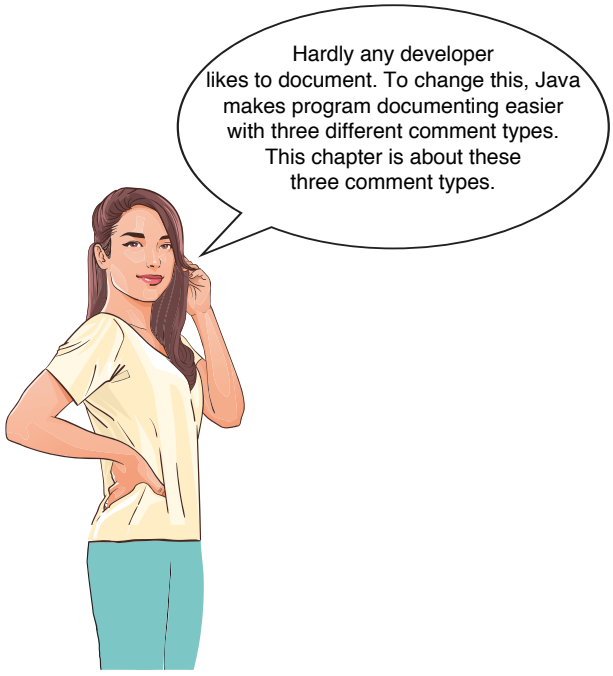


Figure 18.1: Java facilitates documenting a program with three comment types.

18.2 Overview

Java, like many things from the C/C++ era, has gently revolutionized documentation inside and outside a program. Therefore there are three different comment types (Table 18.1).

Table 18.1: This table shows which documentation types Java offers.

Document type	Description
Inline comment	Comments a single line
Block comment	Comments a block
Documentation comment	JavaDoc can be generated from this comment

18.3 Line Comments

This comment type is introduced by double slashes, which mark the rest of the line as a comment. Thus, they refer to only a single line at a time (Figure 18.2).

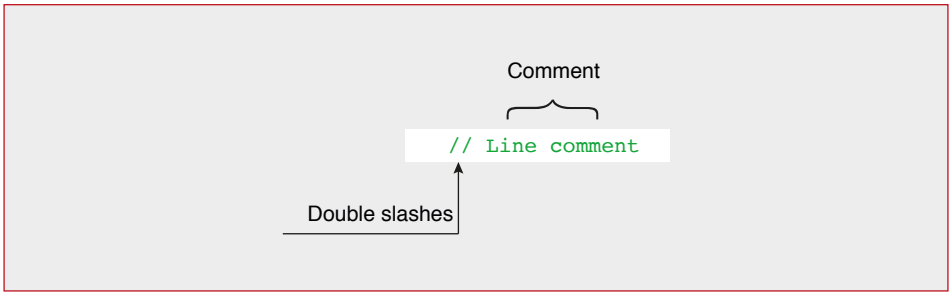


Figure 18.2: For line comments, from the slashes the line remainder is a comment.

Listing 18.1 gives you two examples for a line comment. As you can see, the comment can be placed before the statement to which it belongs or next to it.

Listing 18.1: An example how to use a line comment.

```
1 (...)
2 // Line comment before a statement:
3 statements // Line comment after a statement
4 (...)
```

18.4 Block Comments

In contrast to line-related comments, block-related comments (block comments) can be used to hide large parts of the source code for the compiler and mark them as comments. As in the C/C++ programming language, they are started with a slash followed by an asterisk and end in the reverse order (Figure 18.3).

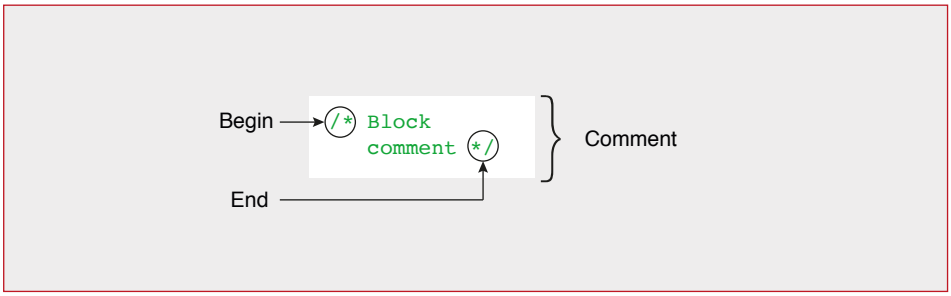


Figure 18.3: A block comment can span several lines.

For this comment, Listing 18.2 shows an example.

Listing 18.2: An example how to use a block comment.

```
1 /* Block comment
2    spans
3    over
4    several lines */
```

However, the block comment can also be used to insert comments in the middle of the source code.

Listing 18.3: A block comment can also be used to comment a single line.

```
1 (...)
2 Statements
3 /* Block comment refers to one line only */
4 Statements
5 (...)
```

18.5 Documentation Comments

This interesting comment type is used to create HTML documents from comments inserted in the source code. These comments can also span several lines, ending like the section-related comments, but starting with an additional asterisk (18.4).

Listing 18.4 shows an application based on a comment from the program example *Course StatisticsApp* at the end of this book.

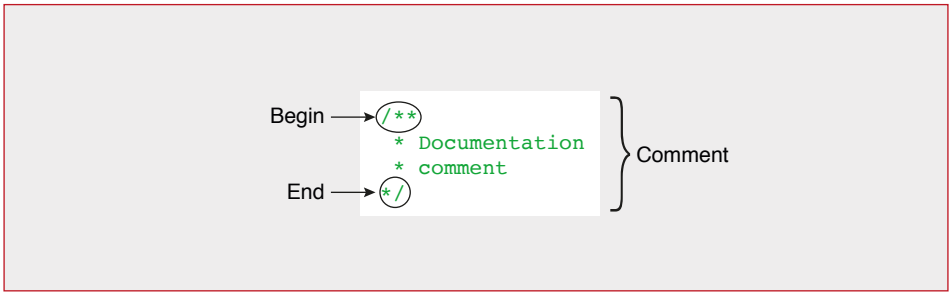


Figure 18.4: JavaDoc comments are used to create a documentation.

Listing 18.4: An example of a JavaDoc comment with authors and version.

```
5  /**
6   * Project: CourseStatisticsApp
7   * Description: Book "Getting started with Java using Eclipse".
8   * @copyright (c) 2023 by Bernhard Steppan
9   * @author Bernhard Steppan
10  * @version 1.0
11  */
```

The @ character introduces reserved keywords. Table 18.2 gives an overview of the most important tags and parameters.

Table 18.2: An overview of the most important tags and parameters.

Tag and parameters	Description	Usage
@author name	author name	class, interface
@version version	version entry (at most once per class or interface)	class, interface
@since jdk-version	Since when the functionality exists	class, interface, object variable, method
@param name description	description of the parameter of a method	method
@return description	description of the return value of a method	method

Eclipse generates Java documentation from the tags. If that is not enough, there are also Java tools that can generate more sophisticated documentation from the documentation comments fully automatically.

18.6 Summary

Java offers three different comment types: Line-related comments are introduced by two slashes. They apply from the moment you insert them until the rest of the line. They are good for single-line comments like subheadings. If comments are to extend over several

lines, a section-related comment is used. It starts with a slash followed by an asterisk, and ends with an asterisk followed by a slash.

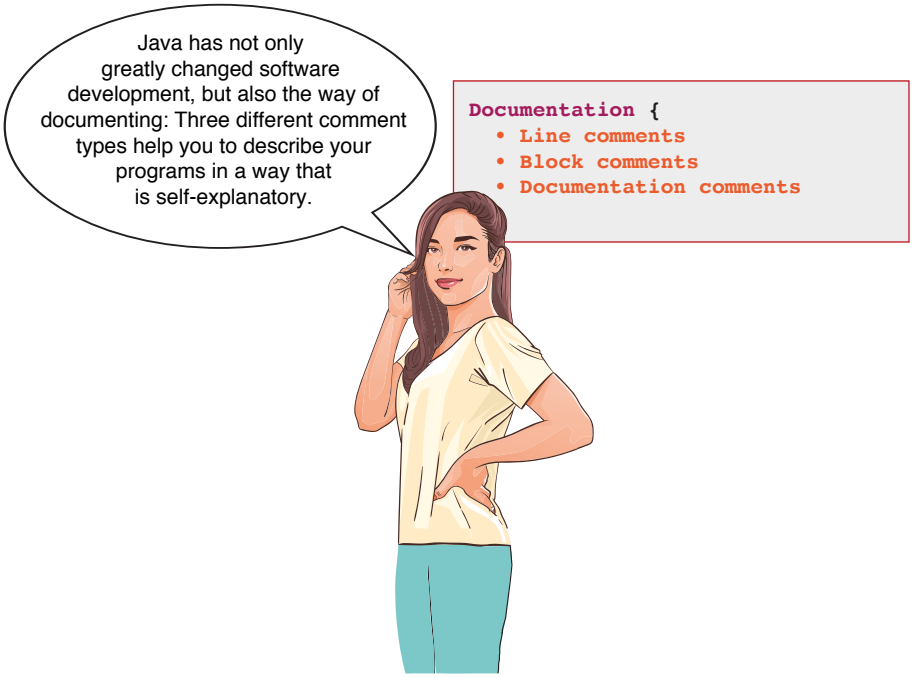


Figure 18.5: Java makes it easy to document a program with three types of comments.

The third comment type that Java provides is Javadoc. Behind this term is a special type of documentation that the development environment can read to provide the developer with information about a class or method. There are special tools that can generate static web pages from Javadoc, as you know from the official Java documentation.

18.7 Literature

Javadoc: *https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html*

18.8 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/documentation

18.9 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/documentation
There you will also find the solutions to the exercises.

19 Annotations

19.1 Introduction

Since Java 5, annotations have been part of the Java programming language. They are annotations that you can add to packages, classes, interfaces, methods and variables. Although they are not really a topic for programming beginners, you should at least know what annotations are and what they are used for. This chapter gives you a first overview.

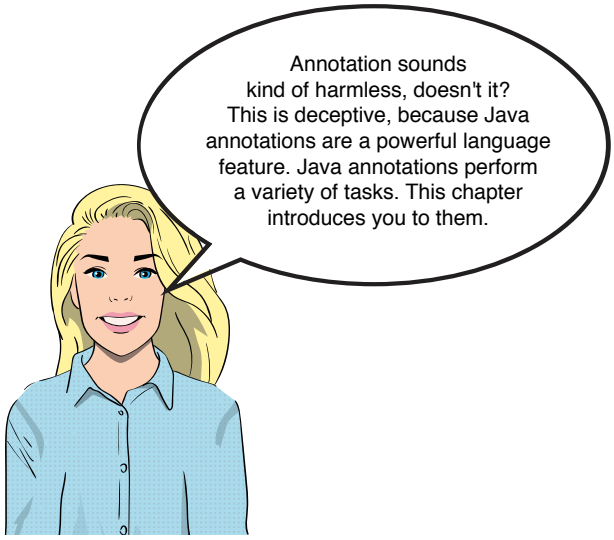


Figure 19.1: Java annotations perform a variety of tasks in Java.

19.2 Overview

19.2.1 Annotation Purposes

Annotations are annotations that you can use to mark up any language elements such as packages, classes, interfaces, methods, and variables. You can add annotations to your own classes, but you will also use classes that already contain annotations. You can use annotations to do the following:

- Tell the compiler not to display warnings or mark classes as obsolete.
- Make changes at compile time, for example, generate source code
- Modify or check the behavior of a program at runtime.

19.2.2 Annotation Types

In order to achieve the previously mentioned tasks, there are the different types of annotations listed in Table 19.1.

Table 19.1: This table gives you an overview of annotation types.

Annotation type	description
Compiler control annotations	This annotation type is used to control the compiler, for example to suppress warnings
Meta-annotations	This type of annotations are intended to define other annotations
Framework-specific annotations	Many frameworks use their own annotations, for example to store data permanently or for configurations

Annotations for compiler control are, for example, able to suppress warnings that the development environment would normally issue. This is useful if you have identified the reason for the warnings, but want to reduce the number of warnings in the project to avoid confusion.

You use meta-annotations only if you want to develop your own annotations. Annotations are defined like classes in a separate file. For this purpose, certain specifications must be fulfilled, for which a developer uses the so-called meta-annotations. With them, the developer of an annotation specifies, for example, to which language element this annotation can be applied.

You will encounter annotations for configuration in certain frameworks. These are used to specify certain base values of the framework. Closely related to this are framework-specific annotations, which you might use to specify that a class corresponds to an entity of a database, for example.

19.2.3 Predefined Annotations

Standard Annotations

You will not normally program annotations yourself. Instead, you will resort to those that are already part of the Java language as standard annotations. Table 19.2 shows an overview of the predefined standard annotations. Especially the first three annotations are frequently encountered in the source code of Java programs.

Table 19.2: This table presents an overview of the predefined standard annotations.

Annotation	Description
@Deprecated	Marks a class, interface, method, or attribute as deprecated or obsolete
@SuppressWarnings	Suppresses warnings and error messages
@Override	Marks a method as overridden. The compiler checks whether a method that can be overridden even exists in the base class and issues an error if it does not.
@SafeVarargs	Assures that subsequent method does not perform unsafe operations on its varargs parameter
@FunctionalInterface	Specifies that the interface should be a functional interface according to the language definition

Meta-Annotations

If the previously mentioned standard annotations are not enough for you, you have to program your own annotations. To make this easier, Java provides so-called meta annotations. These are only needed to define your own annotations (Table 19.3).

Table 19.3: This table presents an overview of the predefined meta-annotations.

Annotation	Description
@Retention	Specifies how the annotation marked with it will be stored (this can be in source code, bytecode or at runtime)
@Documented	Indicates that the use of this annotation should be documented (by default, the use of annotations does not need to be documented)
@Target	Specifies to which language elements this annotation can be applied, for example »normal« methods, constructions, local variables, packages or all language elements
@Inherited	Indicates that this annotation can be inherited from the superclass (applies only to annotations that refer to classes)
@Repeatable	Indicates that this annotation can be used multiple times in the same place to mark up the source code

19.2.4 Use of Annotations

Annotations are placed *before* the language element they refer to. They begin with the @ character, which is followed by the name of the annotation. In the simplest case, an annotation consists of just an identifier with no parameters. In figure 19.2 you can see an annotation that marks the following method as obsolete.

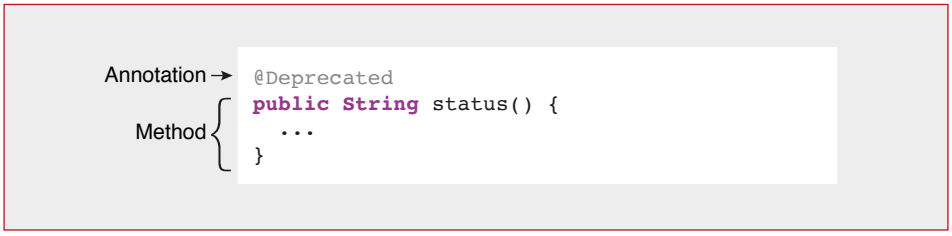


Figure 19.2: Example of a simple annotation without parameters.

Some annotations can or must be called with one or more parameters. Through the parameters one determines, for example, what the annotation refers to. If only one value is possible, the annotation looks like in the Figure 19.3. In this case, the annotation suppresses warnings of unused components of a class. For example, this may be a method that is never called.

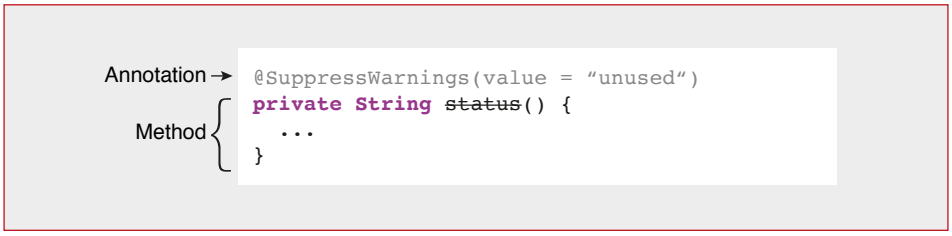


Figure 19.3: An annotation with a parameter as a long form.

If there is only one type of parameter, then the assignment using *value* is not strictly necessary. Instead, you can just pass the value as a parameter in quotes. In Figure 19.4 you can see again the annotation *SuppressWarnings*. Since there is only one parameter in this annotation called *value*, this statement can be written in a shortened form.

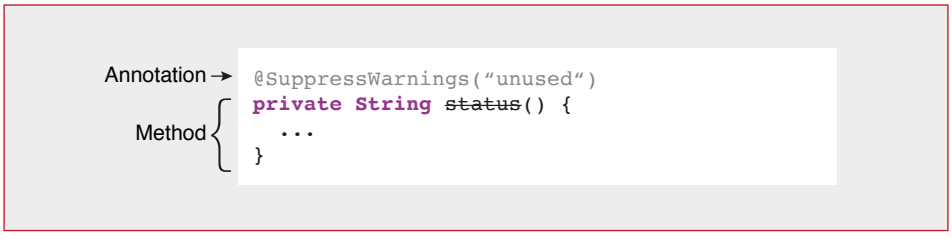


Figure 19.4: An annotation with a parameter as a short form.

If it is possible to specify several differently named parameters, you must specify them as a sequence of assignments separated by commas. An example is again shown by the annotation *Deprecated* in Figure 19.5.

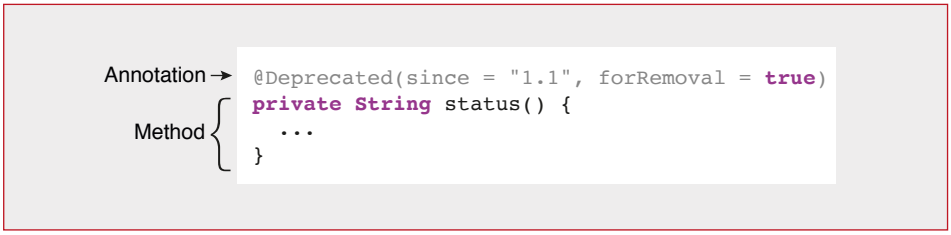


Figure 19.5: Using an annotation with multiple parameters.

The annotation designates the *status()* method as obsolete by the first parameter as of version 1.1. Moreover, by setting the second parameter *forRemoval* to *true*, every programmer knows that this method will be removed in the next version of the class. If only one parameter type is possible, there is also a short form (Figure 19.6) if more than one value is passed.

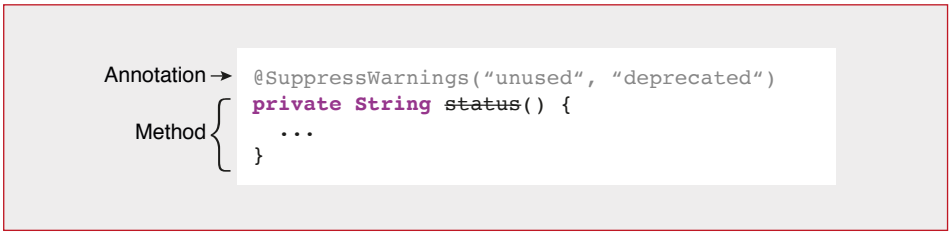


Figure 19.6: An annotation with multiple parameters as a short form.

Some annotations should definitely be annotated so that it is clear to the developer what to use as a replacement in the case of an obsolete class or method, for example (Figure 19.7).

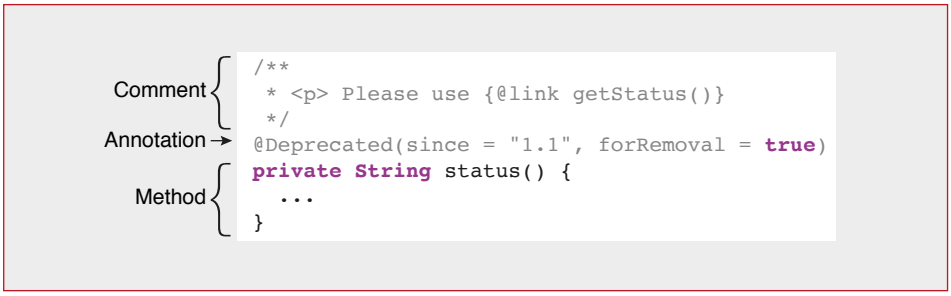


Figure 19.7: An annotation with multiple parameters and a comment.

19.3 Compiler Control Annotations

Finally, I would like to show you the use of the most important annotations for practical Java programming, the annotations for compiler control, with some program examples. At the beginning the annotation »Deprecated« shows you how to mark source code as obsolete. The next example shows you how to suppress warnings in the source code with the annotation »SuppressWarnings«. Finally, the annotation »Override« marks a method that overrides a method of the base class.

19.3.1 Annotations »Deprecated«

This annotation marks certain parts of the source code as deprecated. What for? Imagine that you have programmed a method that has turned out to be not very successful over time and that you therefore want to replace it with a better method. Privately, you would simply delete the method or change it via refactoring. But if you are writing software for other programmers, you cannot simply delete or rename the old method. It may be used by others in the confidence that it will exist the same tomorrow.

How do you still get rid of the blunder in the design of a method as quickly as possible? The solution: You simply mark the old method with the appropriate annotation as *deprecated* – so it is clear that this method is »obsolete«. This way you make it clear that you will soon no longer support the method. The compiler will then warn anyone who uses the class in the next version of the software you ship not to continue using the method.

Without Parameters

How to use this annotation without parameters is shown in Figure 19.8. If you take a closer look at the snippet, you will see that there is a comment before the annotation. The comment gives a hint which method should be used instead of the deprecated one. Such a comment is not mandatory. However, it should be added because otherwise programmers using the class will not know what to use as a replacement for the deprecated method.

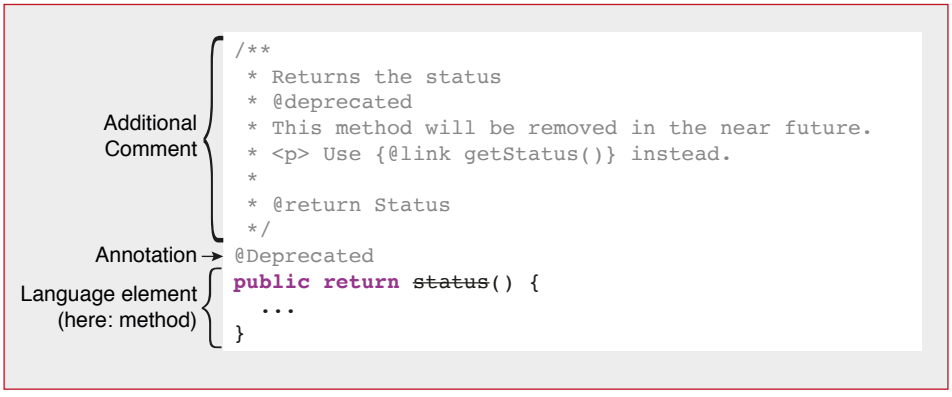


Figure 19.8: Using of the annotation »deprecated« without parameters.

You can also see from the snippet in Figure 19.8 that the name of the method has been crossed out. Of course, if you view the class in a normal text editor, the method will not appear crossed out. This is a feature of Java development environments such as Eclipse, whose editor evaluates the annotation and displays the language element crossed out. This visually highlights a development environment that urgent action is needed.

With Parameters

Java 9 improved the Deprecated annotation by adding the *since* and *forRemoval* parameters. The *since* parameter allows to specify a version of the class from when an element was marked as deprecated. The boolean parameter *forRemoval* should be set to *true* whenever the element is removed in the next version. If you do not specify a parameter for this, *forRemoval* will have the value *false* by default.

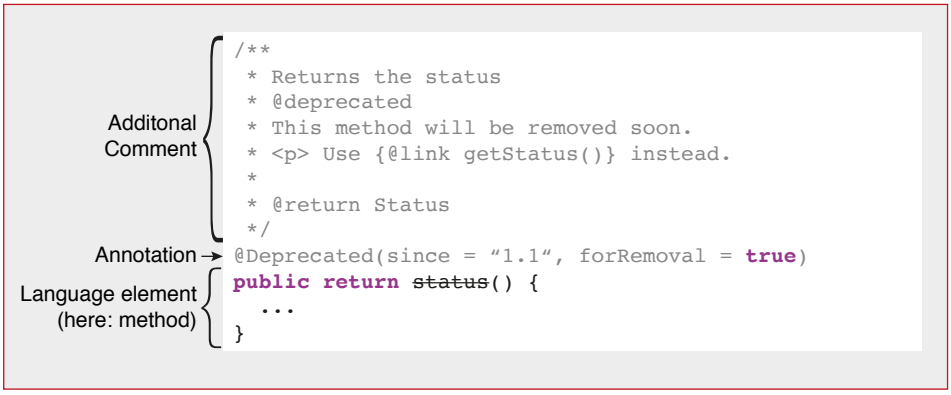


Figure 19.9: Using of the annotation »Deprecated« with two additional parameters.

The listing 19.1 shows a code example for a method marked as *deprecated* with two parameters. The class is not marked as *deprecated* here.

Listing 19.1: The method »status()« is »Deprecated«.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  public class DeprecatedMethod {
6
7    /**
8     * This method demonstrates the use of the annotation
9     * "deprecated" with parameters on a class
10    * @deprecated
11    * This method will be replaced soon.
12    * <p> Use {@link getStatus()} instead.
13    * @returnStatus
14    */
15   @Deprecated(since = "1.1", forRemoval = true)
```

```
16    public String status() {
17      return "I am a deprecated method.";
18    }
19
20    public String getStatus() {
21      return "I am the replacement.";
22    }
23  }
```

The situation is different if you have marked the entire class as obsolete (Listing 19.2). This marks every element of the class as deprecated.

Listing 19.2: The class »DeprecatedClass« is »Deprecated«.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  /**
6   * This class demonstrates the use of the annotation
7   * "deprecated" with parameters on a class
8   * @deprecated
9   * This class will be replaced soon.
10  * <p> Use {@link NewClass} instead.
11  */
12 @Deprecated(since = "1.1", forRemoval = true)
13 public class DeprecatedClass {
14
15   public String status() {
16     return "I am a deprecated class.";
17   }
18 }
```

You can see how the Eclipse development environment reacts to the two classes in Figure 19.10. On line 10, the program creates a new object based on the *deprecatedmethod* class. As you can see, using the class does not generate a warning yet. The situation is different in line 13. Here you can see how Eclipse reacts to the use of the *status()* method with a warning.

Starting at line 16, the program uses the *deprecatedclass* class. Here, Eclipse already reacts with a warning when calling the constructor in line 16, which is repeated when the program tries to call the *status()* method in line 19. This means that if a class is marked with *deprecated*, all elements are also *deprecated*.

19.3.2 Annotations »SuppressWarnings«

Unlike the deprecated annotation, the annotation *SuppressWarnings* annotation does not generate a warning, but suppresses it. This is useful if you understand the warning and have planned countermeasures or cannot do anything about the cause. Eclipse does allow to suppress warnings. But this only works for a project or a workspace for all warnings of

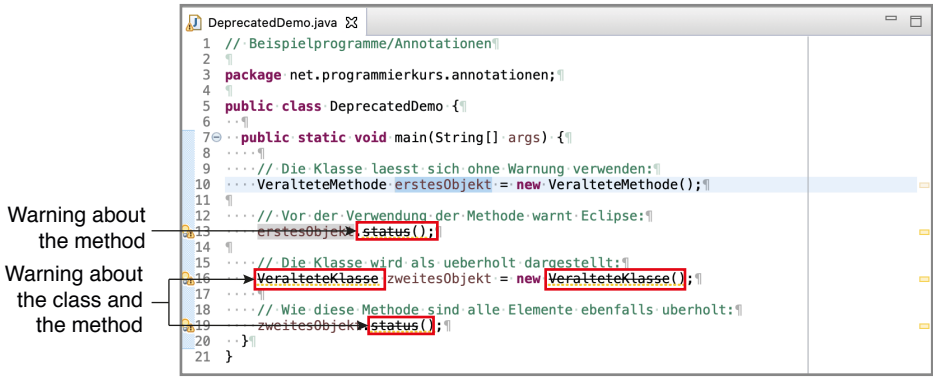


Figure 19.10: Eclipse warns not to use the deprecated method and class.

one kind, which would not be a solution. The annotation *SuppressWarnings* , on the other hand, can be used to specifically suppress each individual warning. Figure 19.11) shows how to use this annotation.

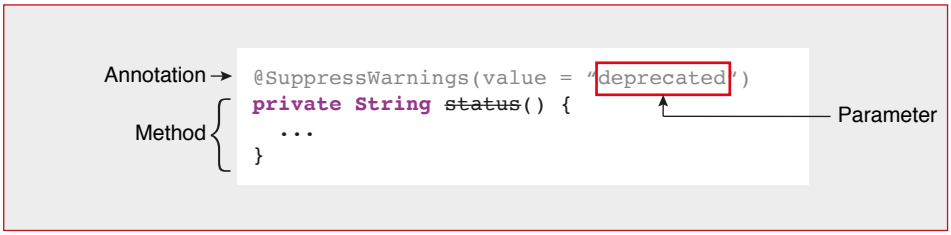


Figure 19.11: The annotation »SuppressWarnings« as a long form.

As mentioned at the beginning, there is also a short form notation for this (Figure 19.12). The shorthand notation can be used because the annotation has only one kind of parameter called *value*, so it is clear what the passed value refers to.

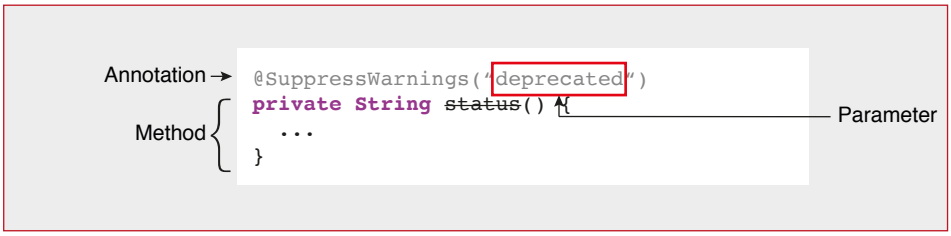


Figure 19.12: The annotation »SuppressWarnings« as a short form.

If you want to suppress multiple warnings at once in one code block, you can pass them one after another in a parameter list. Now, of course, the question is what values there are for the *value* parameter. There are a lot of values: as many values as there are different warnings. So the values depend a lot on the development environment and the Java version. I have compiled the most important values for you in table 19.4.

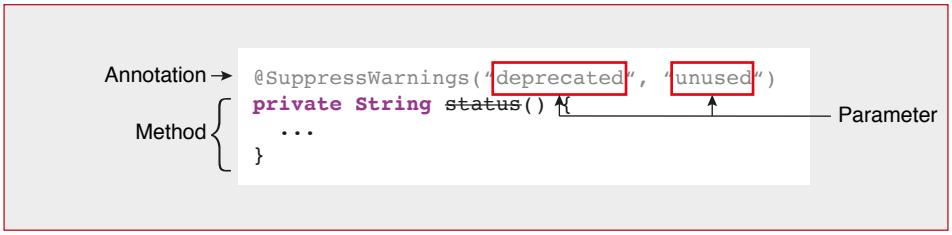


Figure 19.13: The annotation »SuppressWarnings« as a short form with multiple parameters.

Use the *all* value to suppress all warnings for the element following the annotation. The *deprecation* value, on the other hand, suppresses only warnings about an obsolete language element. This must not have already been marked with *removal*. If it is marked with *removal*, you need a parameter with the same name to suppress the warning.

Table 19.4: The most important parameter values of the »SuppressWarnings« annotation.

Parameters	description
all	Suppresses all warnings
deprecation	Suppresses a deprecation warning (without removal)
removal	Suppresses a deprecation warning (with removal)
serial	Suppresses a warning that a serial number is missing
unchecked	Suppresses a warning that a type check is missing
unused	Suppress a warning that a code section is not executed

Classes that you can write to disk (serialize) must have a serial number. If you have not added a number to such a class and want to suppress the warning, use the parameter *serial*. Warnings about non-existent type checking can be bypassed with the *unchecked* parameter. The last parameter from the table is unused and hides warnings that occur when a section of code is not executed.

By the way, you don't have to remember these parameter values. The Eclipse"=development environment allows you to automatically insert the appropriate annotations for each warning. Let's take a look at this with an example. The class *SuppressWarningsUnusedDemo* contains an unused method: the method *status()* is never called within the program (Listing 19.3).

Listing 19.3: A class with unused method »status«.

```
1  // sample programs/annotations
2
3  package net.programmingcourse.annotations;
4
5  public class SuppressWarningsUnusedDemo {
6
7    private void status() {
8      System.out.println("I will never be called.");
9    }
```

```
10
11    public static void main(String[] args) {
12      System.out.println("Demo for parameter 'unused'");
13    }
14  }
```

The Eclipse development environment may determine that the method is never called because, first, it is declared as *private* and, second, a section of code that uses the method is missing. Therefore, Eclipse displays a warning. This is very useful if you accidentally forget to delete the unused method.

However, imagine that you are developing this method only to add it to your program later. Then the warning would disturb you in your work. In this case, move the mouse pointer over the warning and wait until the editor's programming help appears. Then choose from the options ADD @SUPPRESSWARNINGS 'UNUSED' TO 'STATUS()' (Figure 19.14).

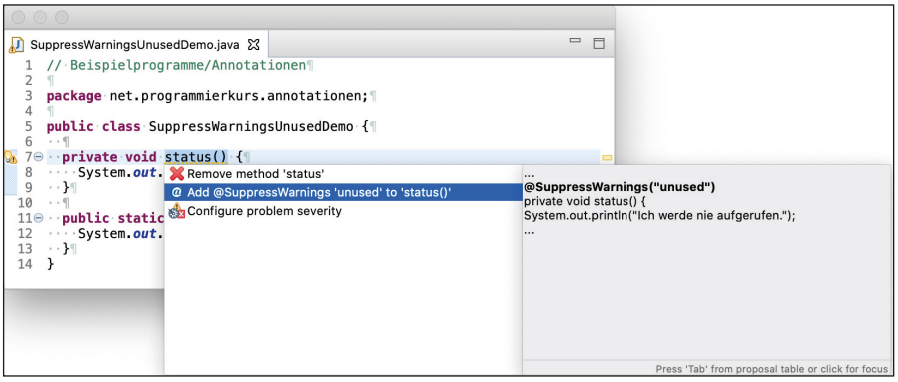


Figure 19.14: Annotations are used to mark up the source code.

As you can see, you don't need to remember the parameter values. Eclipse provides you with a suitable hint via the programming help, which annotation with which parameter values can be inserted at this point to suppress the warning. After that, the class should look like Listing 19.4 shows.

Listing 19.4: The warning »unsed« is suppressed.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  public class SuppressWarningsUnusedDemo {
6
7    @SuppressWarnings("unused")
8    private void status() {
9      System.out.println("I am never called.");
10   }
11
12   public static void main(String[] args) {
13     System.out.println("Demo for the parameter 'unused'");
```

```
14    }
15  }
```

Let's conclude this section by looking at the different effect of the SuppressWarnings annotation on classes that have been marked as *deprecated*, that is, obsolete. Here, there is an option to mark the class with *forRemoval* or not. Let's first take a class in version 1.1 without this parameter (Listing 19.5).

Listing 19.5: A class with the unused method »status«.

```
1  // sample programs/annotations
2
3  package net.programmingcourse.annotations;
4
5  /**
6   * This class demonstrates the use of the annotation
7   * "deprecated" with parameters on a class.
8   * This class will be replaced soon.
9   * <p> Use {@link NewClass} instead.
10  * @author Bernhard Steppan
11  * @version 1.1
12  * @deprecated
13  */
14 @Deprecated(since = "1.1")
15 public class DeprecatedClass {
16
17   public void status() {
18     System.out.println("I am a deprecated class.");
19   }
20 }
```

To suppress the warning, it is sufficient to annotate *SuppressWarnings* with the parameter value *deprecation*. In this case, the annotation must be placed before the *main()* method so that the warning takes effect for both the call to the constructor on line 11 and the call to the method on line 14 (Listing 19.6).

Listing 19.6: The class warning is suppressed via »deprecation«.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  public class SuppressWarningsDeprecationDemo {
6
7    @SuppressWarnings("deprecation")
8    public static void main(String[] args) {
9
10     // This class is shown as deprecated:
11     DeprecatedClass objekt = new DeprecatedClass();
12
13     // Like this method, all elements are also deprecated:
```

```
14      objekt.status();
15    }
16  }
```

The situation is different if the programmer of the source class has marked it with *forRemoval*. In this case the class will be removed soon. This class in version 1.2 would look like in listing 19.7.

Listing 19.7: This class is marked as »forRemoval«.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  /**
6   * This class demonstrates the use of the annotation
7   * "deprecated" with parameters on a class.
8   * This class will be replaced soon.
9   * <p> Use {@link NewClass} instead.
10  * @author Bernhard Steppan
11  * @version 1.2
12  * @deprecated
13  */
14 @Deprecated(since = "1.1", forRemoval = true)
15 public class DeprecatedClass {
16
17   public void status() {
18     System.out.println("I am a deprecated class.");
19   }
20 }
```

To suppress the warning that triggers the use of this class, it is no longer sufficient to provide *SuppressWarnings* with the parameter value *deprecation*. If you would continue to use the old parameter value, *SuppressWarnings* will stop working and Eclipse will display a warning (Figure 19.15).

The developers of the annotation wanted the newly introduced parameter *forRemoval* to ensure that a warning once suppressed with *deprecated* is not unknowingly hidden. To do this, the deprecated annotation switch must be deliberately flipped from *deprecated* to *removal* (Listing 19.8).

Listing 19.8: The class warning is suppressed via »removal«.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  public class SuppressWarningsRemovalDemo {
6
7    @SuppressWarnings("removal")
8    public static void main(String[] args) {
9
```

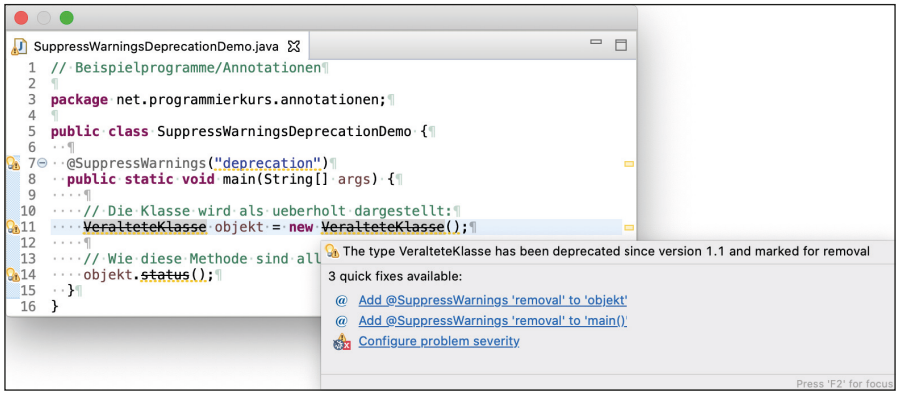


Figure 19.15: The value »deprecation« is no longer sufficient to suppress the warning.

.

```
10      // This class is shown as deprecated:
11      DeprecatedClass objekt = new DeprecatedClass();
12
13      // Like this method, all elements are also deprecated:
14      objekt.status();
15    }
16  }
```

The reason for this procedure is that a class or method that a developer has marked as *deprecated* and »armed« with the *forRemoval* parameter will soon be removed. Programs that continue to use the class or method partout must make a conscious decision to suppress the warning, since these programs will of course no longer be able to be compiled after the class or method is removed. So it is a "last warning" to the users of the class or method to change their program as soon as possible.

19.3.3 Annotation »Override«

The last of the predefined standard annotations I want to introduce to you in this chapter is called the override annotation. You simply place it in front of the method that is to override a method of the base class (Figure 19.16). It is used to avoid errors when overriding methods. As with the previous example, it is again important to understand the safety aspect as the reason for using the annotation.

If you override a method, you want to change the behavior of the base class. It should be self-evident that the underlying method exists at all. But the compiler cannot check this without annotation. If you want the compiler to check whether an override works, you have to use the override annotation. It is a compiler statement: check if a base method of the same name exists and print an error message if not .

As an example again the base class with the two methods *status()* and *getStatus()* shall be used. In our case study the developer of this class should notice that the method *status()* in version 1.1 of this class is nonsensical. He then decides to replace it with the *getStatus()*

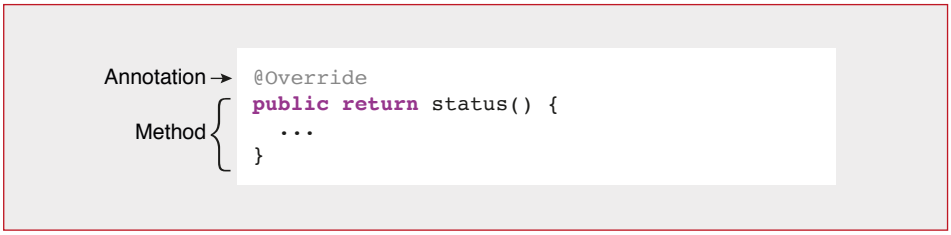


Figure 19.16: The annotation »Override« ensures that the base method exists.

method. Of course, before version 1.1 ships, he marks the *status()* method as *deprecated* with the parameter value *forRemoval* = *true* to warn other developers using the class of the impending change (Listing 19.9).

Listing 19.9: The base class with the two methods »status()« and »getStatus()«

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  /**
6   * Demo for a base class
7   * @author Bernhard Steppan
8   * @version 1.1
9   */
10 public class BaseClass {
11
12   /**
13    * Please replace with getStatus()
14    * Verwenden Sie {@link getStatus()} stattdessen.
15    */
16   @Deprecated(since = "1.1", forRemoval=true)
17   public void status() {
18     System.out.println("I am the deprecated base method 'status()'.");
19   }
20
21   public void getStatus() {
22     System.out.println("I am the replacement for the method 'status().");
23   }
24 }
```

The class is to be derived from a class, with the new method *status()* overriding the base class method of the same name (Figure 19.10). Note that the Deprecated annotation does not warn about overwriting the *status()* method at this point. There can be no warning because derived class does not use this method directly, but only builds on it.

Listing 19.10: The class derived from the base class.

```
1  // Examples/Annotations
2
3  package net.programmingcourse.annotations;
4
5  /**
6   * Demo for a derived class
7   * @author Bernhard Steppan
8   * @version 1.1
9   */
10 public class DerivedClass extends BaseClass {
11
12   public void status() {
13     System.out.println("I override the base method.");
14   }
15 }
```

What would happen if the base class developer removed the method *status()* in the next version? The method would then look like in Listing 19.11.

Listing 19.11: The base class without status method.

```
1  // sample programs/annotations
2
3  package net.programming.annotations;
4
5  /**
6   * Demo for a base class
7   * @author Bernhard Steppan
8   * @version 1.2
9   */
10 public class BaseClass {
11
12   public void getStatus() {
13     System.out.println("I am the replacement for the method 'status().");
14   }
15 }
```

The answer is that nothing happens. How could the compiler even warn? It cannot suspect that the programmer's original intention was to override the *status()* method. However, the consequences may be serious, since the purpose of overwriting is to change the behavior of the class.

Therefore, whenever you override a method, you should add the annotation *Override* annotation in the derived class. Through it, Eclipse checks if the base method is missing and, if so, displays an error that prevents the program from running (Figure 19.17).

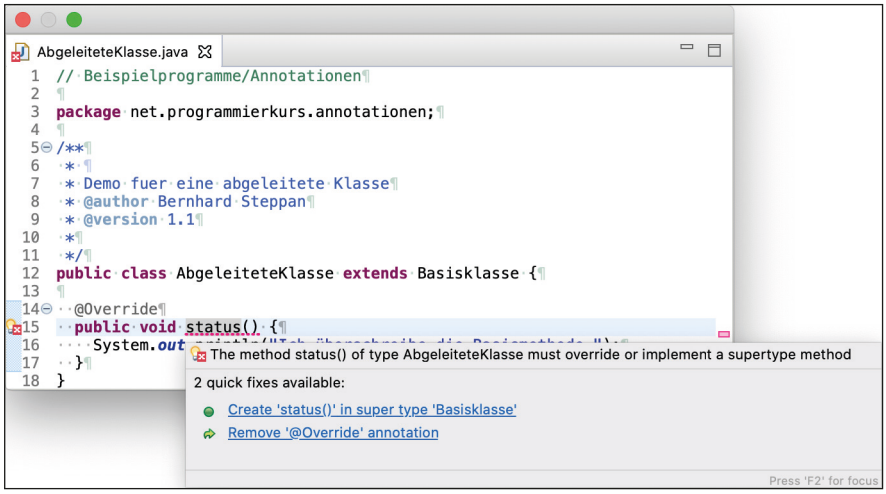


Figure 19.17: Eclipse now shows an error because the base method is missing.

19.4 Summary

Annotations are used to mark up Java language elements. You can use them to add annotations to packages, classes, interfaces, methods and variables. They are mainly used to make the compiler display warnings in case of deprecated classes or methods, for example.

19.5 Literature

Annotations: *https://docs.oracle.com/javase/tutorial/java/annotations*

19.6 Tutorial

Please continue the chapter now with the online tutorial with Eclipse:
https://www.programmingcourse.net/books/java_with_eclipse/annotations

19.7 Exercises

When you have finished the tutorial, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/annotations
There you will also find the solutions to the exercises.

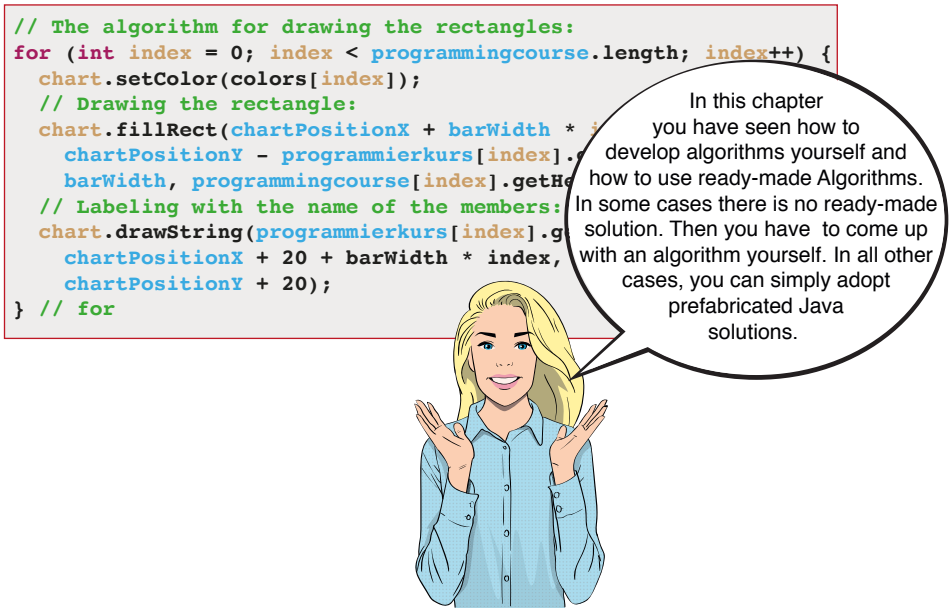


Figure 19.18: Annotations are used to mark up the source code.

PART III

Java Technology

The Java technology differs from pure programming languages such as C/C++ in that the Java inventors did not »just« provide a language definition. In addition to its language properties, Java also has a technology character. The technology is the part of the overall concept that ensures the high portability of Java programs. This includes the virtual machine and the intermediate format called bytecode.

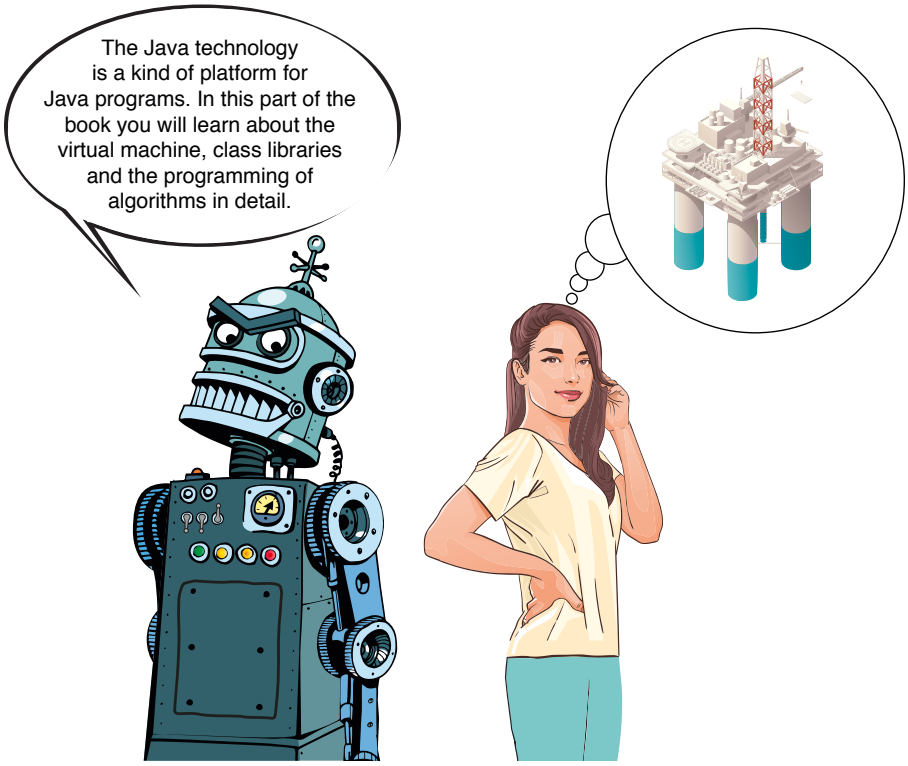


Figure 19.19: The Java technology Java is the platform for Java programs.

In this part of the book, you will learn about the different components of the Java tech-nology. The book part starts with the chapter »Development Processes«, which explains compiler and debugger in more detail. Then it moves on to the core component of the Java technology, the »Runtime Environment« that runs your Java programs. The chapters »Class Libraries«, »Rules« and »Algorithms« complete the tour of the Java technology.

20 Development Processes

20.1 Introduction

This chapter highlights the processes involved in the development of Java programs. The scope extends from the planning, programming and testing of a new program to the delivery of the finished Java software.

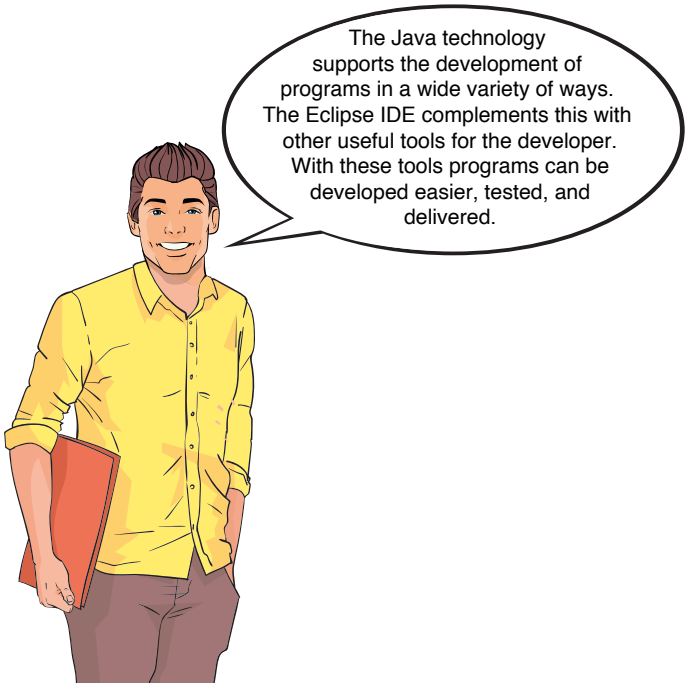


Figure 20.1: Eclipse supports software development through various tools.

In this chapter, you will learn how to derive classes from requirements, design a program, and debug using the Eclipse debugger. The chapter concludes with how to package and deliver a program using Eclipse.

20.2 Overview

Development processes are recurring workflows in software development. To get a better overview of the different workflows, the overall process is divided into related subprocesses. Some companies divide the total process into three, others into four sections.

In order not to go beyond the scope of this book, I would like to limit myself to a three-part process, which is sufficiently precise for most projects. The overall process then consists of the following parts (phases): Planning Phase, Design Phase, and Operations Phase (Figure 20.2).

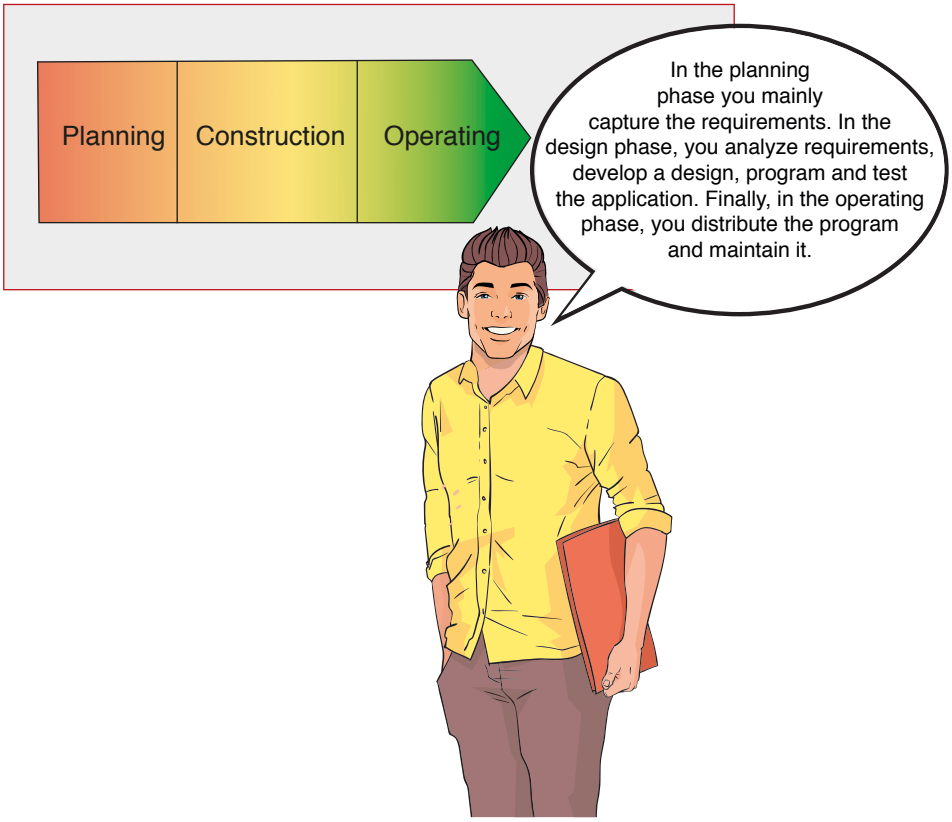


Figure 20.2: The phases of software development.

In the planning phase, the focus is on planning what exactly is to be developed. In the design phase, the focus is on developing (programming) the software product. And in the operation phase, parts of the product or the finished product are installed and then operated. At first glance, the overall process appears to be completely straightforward. But this is a fallacy.

20.2.1 Correlation Between Phases And Activities

In an agile approach, phases are divided into sections of two to three weeks – the so-called sprints. In each sprint, the activities »Requirements Capture«, »Object-Oriented Analysis And Design«, »Implementation« and »Test« run in sequence in software development. How complicated the overall process is becomes clear if you take a closer look at the individual phases (Figure 20.3).

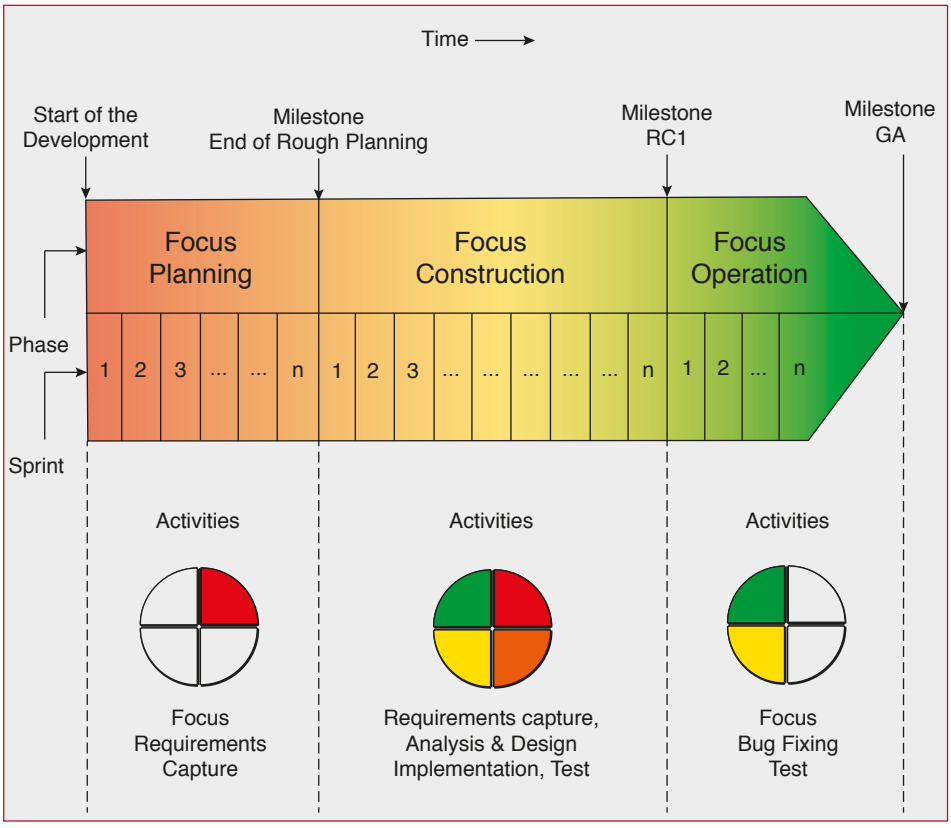


Figure 20.3: Correlation between phases and activities.

In the figure, you can see the individual sprints in the middle. They run from sprint 1 to n in the planning phase, in the construction phase and in the operation phase as well. From left to right, the software product becomes more and more complete through the various activities within the sprints (Requirements Capture, Analysis & Design, Implementation and Test) until delivery and thus operation is reached. Before delivery, the customer is provided with a so-called release candidate (RC), after which the software is »generally available« (GA), i.e. available as a product.

20.2.2 Activities

In each phase, the team engages in a number of different activities:

- Requirements gathering: a team member discusses new requirements with the customer.
- Analysis & Design: An analyst analyzes existing requirements and puts the pieces into a meaningful program design.
- Implementation: Meanwhile, a developer is already programming the software using a rough design.
- Test: Another team member checks whether completed program parts function as desired. If the test is successful in the opinion of the team, a team member compiles the completed software at the end of each sprint.

Afterwards one shows the customer in a review at the end of a Sprints the already finished software. The customer can then accept or reject it (Figure 20.4).

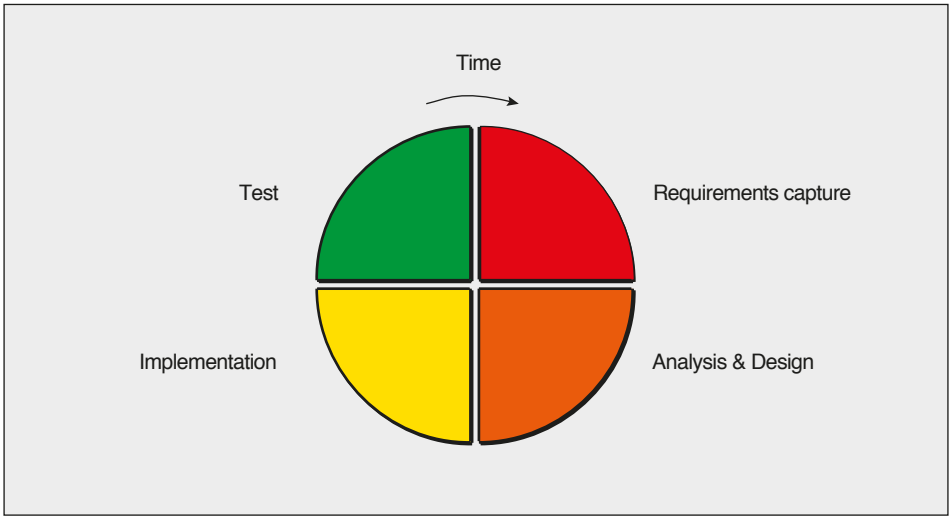


Figure 20.4: The sequence of activities during software development

The processes do not run in a straight line, but rather in a circle, hence the way they are represented. All activities must be run through x times until the overall product is completed. If you apply an agile approach to Java, you first start by recording the requirements in a sprint. Then you derive Java classes and bring them into a meaningful program design. Then the actual Java programming and testing of the Java program begin. These circular processes continue until the project is completed.

The *planning*, *construction*, and *operation* phases summarize certain activities. For example, in the planning phase, a team is mainly engaged in planning (hence the name of the phase). This means that the *focus* of this phase is on requirements gathering. However, it may also be the case that Java code is already being written in the planning phase, for example to complete a prototype.

It is the same with the construction phase. Here, the focus is on analyzing the requirements, finding Java classes, programming Java code and testing this code. But it may also be that new requirements are discovered in discussions with the customer. Then you have to estimate again how long it will take to incorporate these requirements. Afterwards, one converts the changed requirements into Java code, then tests the program and presents it to the customer again.

20.2.3 Tools

A good developer must not only know and master this complicated overall process. He must also select development tools and learn how to use them. The productivity of the developer depends on the selection and integration of appropriate tools. In addition to an agile development process and well-trained developers, good tools and class libraries undoubtedly lead to less stress and noticeably lower development costs.

Regarding class libraries, this part of the book comes in chapter 22, »Class Libraries«. As far as tools are concerned, it is quite sufficient to use the already installed Eclipse now. You should have the Eclipse development environment installed so that you can perform the exercises in this chapter. For a description and installation instructions, see chapter 4, »Development Environment«.

20.3 Planning Phase

In the planning phase, a contract is created between the software developer and the client (customer or end user). This contract states, among other things, what is to be produced, how expensive the entire project may become and what risks the project entails. The contact person for the software developer is, for example, an end user, a business manager, the project or team leader, the chief IT designer or chief IT architect. From one of these contacts, the software developer will learn what tasks he or she will be asked to perform.

20.3.1 Order Clarification

On the one hand, mission clarification causes much of the difficulty in developing the software because end users and software developers do not speak the same language. Therefore, it is necessary to determine exactly what the client wants so that the risk of developing something wrong is reduced. Certain techniques such as an agile approach help with this.

20.3.2 Requirements Capture

The following example: Our programming class with Anna, Julia, Florian and Lukas is supposed to develop a program as part of their internship that displays the students of their university with the courses and electives they have taken. In discussions with a lecturer

Silvia Lenz Julia tries to find out what the content of the internship work should be (Figure 20.5).

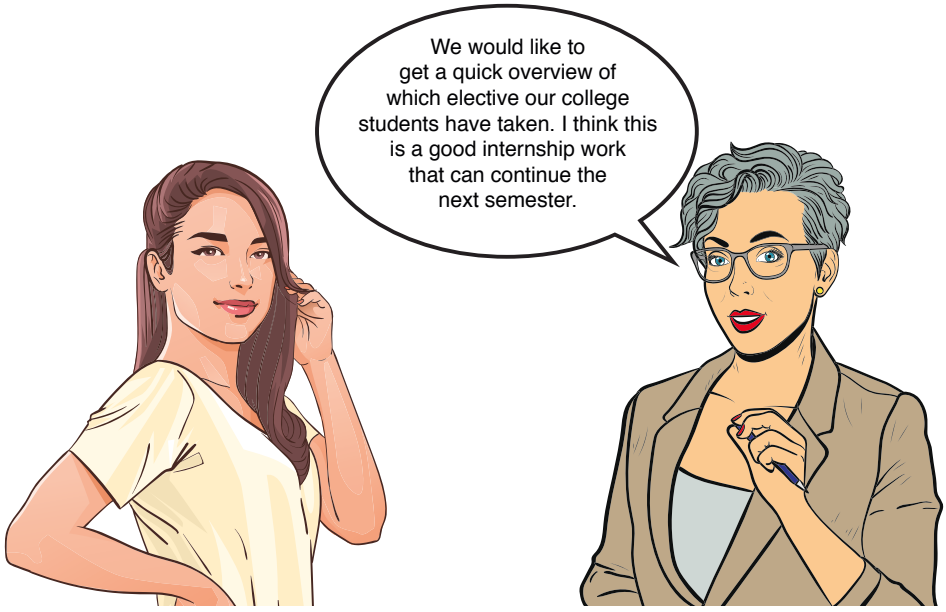


Figure 20.5: Julia clarifies the assignment with instructor Silvia Lenz.

If you take an agile approach, you then write down the requirements in the form of stories. The stories contain, if possible, all the criteria of the client for the deliveries and services of the students of the programming course. In the Stories the requirements are described from user view including all boundary conditions. In practice, the stories are often only a rough guideline, because many end users are not necessarily clear about what they really need.

20.4 Construction Phase

20.4.1 Analysis

After clarifying the details with the instructor, our quartet of students derives a more detailed description in the form of stories. In doing so, they discuss the stories with the instructor, clarifying the university's requirements for the program. Anna develops an initial analysis model from the requirements. Anna's proposal consists of three classes: a class *CourseStatistics*, a class *Student*, and a class *Course*. The course statistics is the main program and displays the statistics of the courses. The class *Student* represents a student with matriculation number, first and last name and the course taken. Finally, a class *course* is necessary to capture the course taken with course number and title (Figure 20.6).

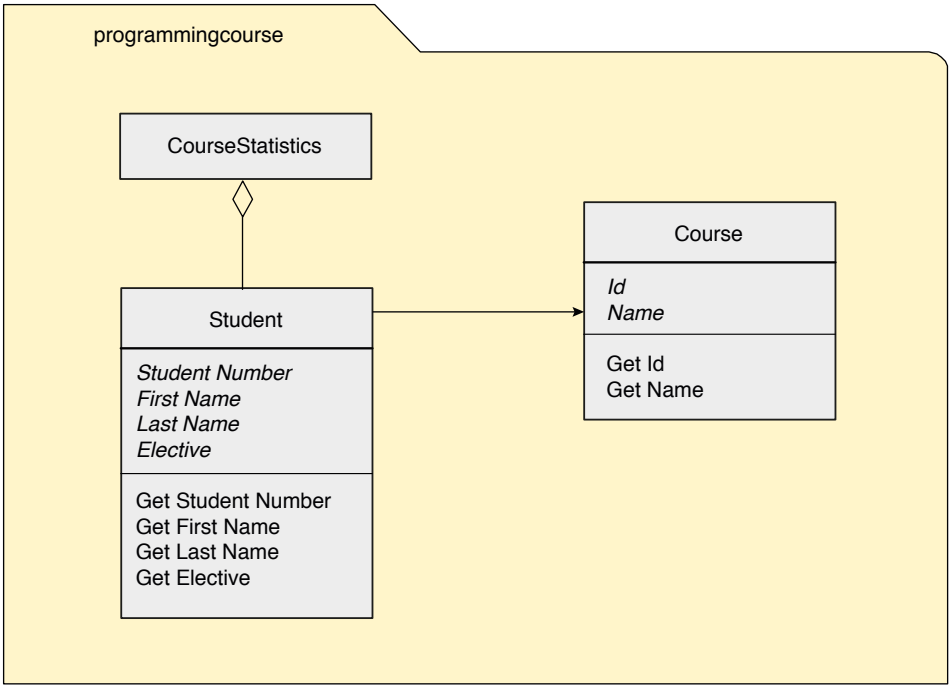


Figure 20.6: Anna has developed a technical analysis model.

From a purely technical point of view, there are admittedly two manifestations of the course at the university: compulsory and elective. Therefore, according to the requirements, Anna has modeled the general class *Course*, which can be used for compulsory and elective subjects if later.

20.4.2 Design

The subject requirements for the student management program must be refined in a technical direction in the next stage of development. The blueprint of a Java program that now emerges is the so-called object-oriented design model. In a Java program, the design model contains, among other things, class models. In object-oriented design, the software designer develops a sensible technical form of the classes that are required for the program.

In Figure 20.7, you can see Lukas' proposal for the design model. Lukas has technically refined Anna's design and added getter methods. He has marked all private attributes with a minus sign, and the public interface methods with a plus sign. Each of the classes listed in this model must be described as Java source code in the next section of the process flow, »Implementation«.

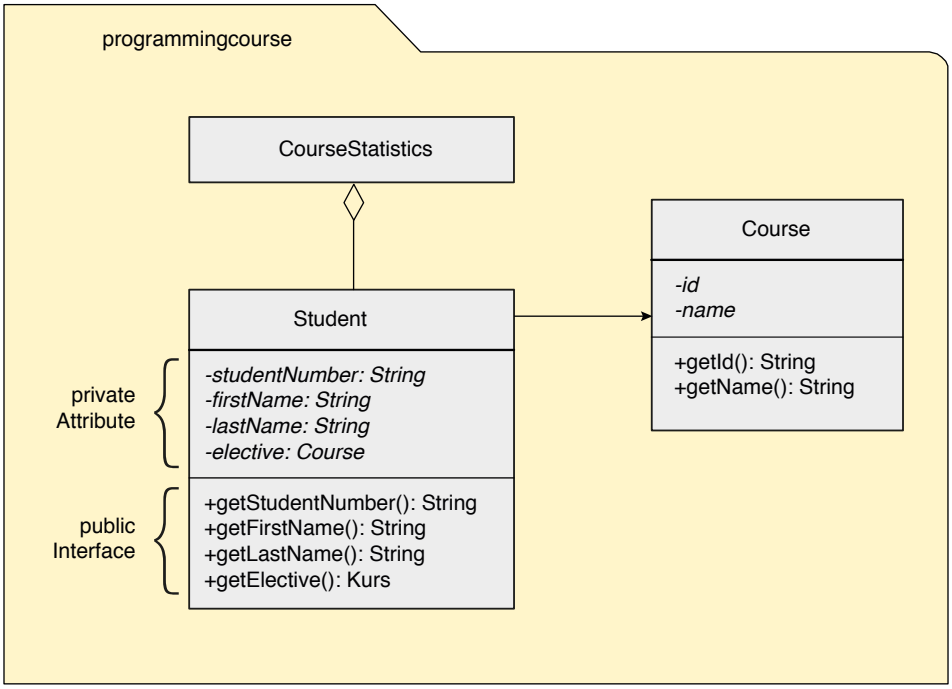


Figure 20.7: Lukas converts Anna's design into a design model.

20.4.3 Implementation

In implementation, a Java developer implements the design of the software using the Java programming language. Which class it makes sense to start the implementation with is a matter of taste. If you want to minimize conflicts in the Eclipse IDE, start with the classes that have the fewest dependencies on other classes.

The class *course* can be implemented easily and without error messages. After that, you would implement the class *Student* and only finally tackle the class *CourseStatistics*. But it also works the other way round: But this way, you provoke Eclipse to create the missing classes automatically. This is exactly what I want to show you on the next pages.

Change Workspace

Now, to start the implementation, start the Eclipse IDE again with the workspace *exercises*.

New Project

To create the classes *Course Statistics*, *Student* and *Course* you need a new Java project again. To do this, start the Eclipse wizard using the FILE → NEW → JAVA PROJECT menu command or the NEW → JAVA PROJECT icon on the far left side of the toolbar. In the dialog that appears, enter the project name *Development_Processes* so that you can assign the project to this chapter.

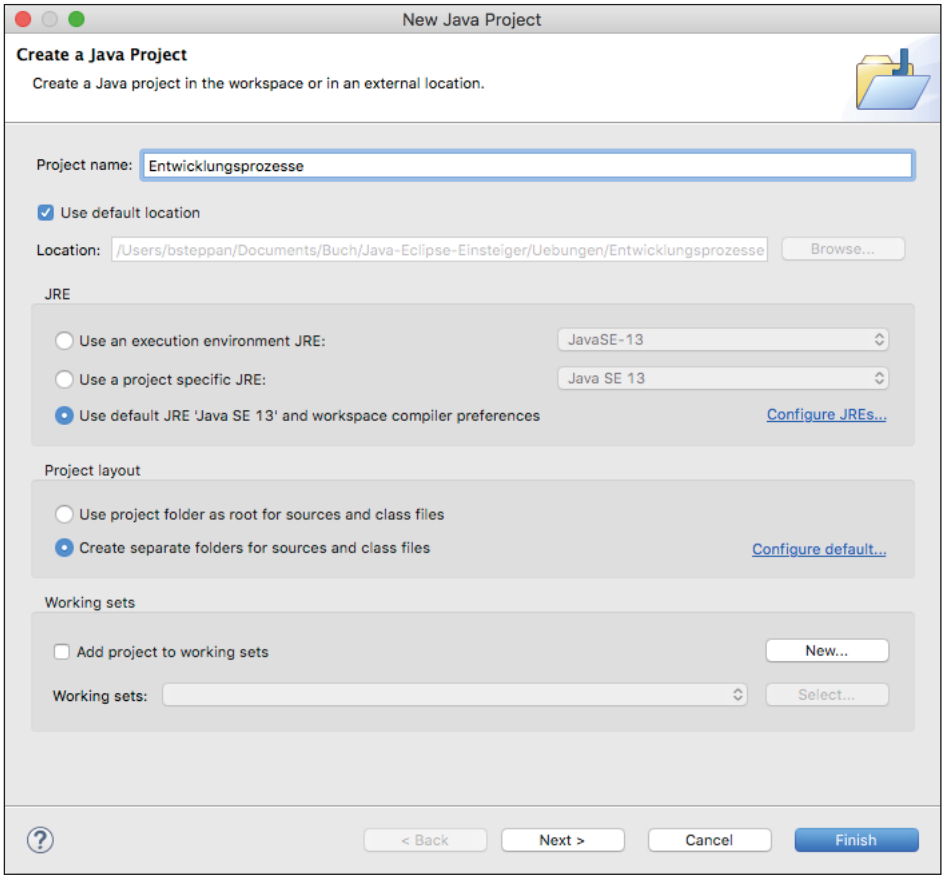


Figure 20.8: Creating the new project »Development_Processes«.

Leave the USE DEFAULT LOCATION option set. It ensures that Eclipse creates the project in the workspace and not in another location on disk. Next, check that the correct Java version is selected under the JRE section. It should be at least Java 13. On the second page of the dialog, deselect the CREATE MODULE-INFO.JAVA FILE option. You can leave the rest of the dialog settings as they are predefined by the development environment. Then create the project by clicking on FINISH.

New Class »CourseStatistics«

You create the new class *CourseStatistics* again using the Eclipse New Class Wizard (Figure 20.9). To do this, invoke the FILE → NEW → CLASS menu command or click the green NEW JAVA CLASS icon in the center of the toolbar. In the dialog that appears, the correct directory for the source code is already filled in with *Development Processes/src*. This is where Eclipse stores the source code of the project. Enter the class name *CourseStatistics* and the package name *ProgrammingCourse*.

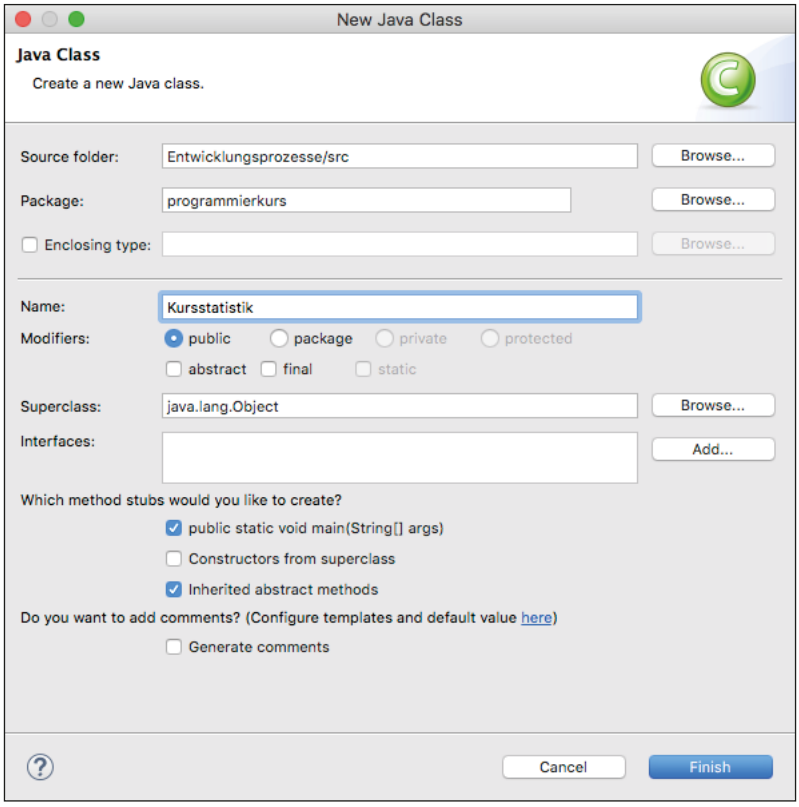


Figure 20.9: Create New Class »CourseStatistics«.

You can leave the option MODIFIER at the default PUBLIC. Eclipse will then create a public class. After that, close the dialog with FINISH. Eclipse has created the empty class *CourseStatistics* in a file named *CourseStatistics.java*. Next, implement the output of the list of students. To do this, use the Eclipse text editor again. Using the Eclipse Editor to do this, add the attributes to the class as seen in Listing 20.1.

Listing 20.1: The first draft of the program »CourseStatistics«.

```
1  package programmingcourse;
2
3  class CourseStatistics {
4
5    public static void main(String[] arguments) {
6
7      Student students[] = new Student[4];
8
9      Course english = new Course(01-SS/2020, "English");
10     Course biology = new Course(02-SS/2020, "Biology");
11     Course history = new Course(03-SS/2020, "History");
12     Course geography = new Course(04-SS/2020, "Geography");
```

```
13
14      students[0] = new Student("100010", "Anna", "Seitz", english);
15      students[1] = new Student("100011", "Julia", "Lippert", biology);
16      students[2] = new Student("100012", "Lukas", "Wittek", history);
17      students[3] = new Student("100013", "Florian", "Krause", geography);
18
19    }
20  }
```

The class consists of an array of course members as well as four electives. Of course, the new implementation results in Eclipse complaining about the two missing classes *Student* and *Course* (Figure 20.10).

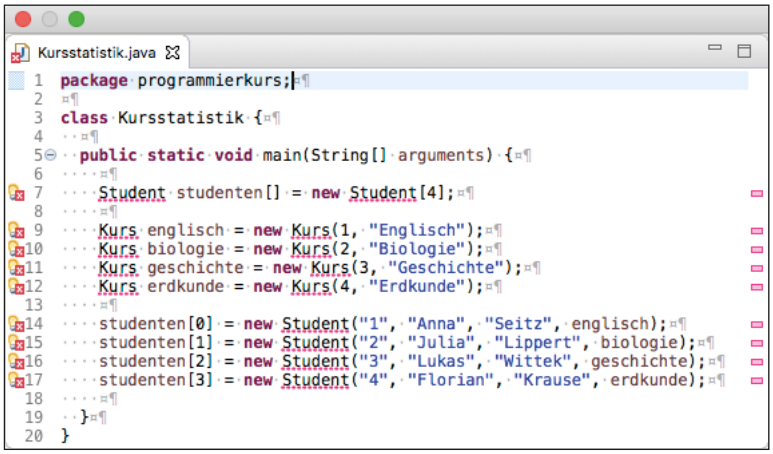


Figure 20.10: Eclipse marks both classes »Course« and »Student« as missing.

Now, to fix the errors, click on the underlined class *Student* and select CREATE CLASS 'STUDENT' from the context menu that appears (Figure 20.11). Eclipse then displays a dialog, which you can exit directly with FINISH, since everything is already filled in perfectly. Eclipse then creates a class shell and opens this new class in the editor on the right side of the development environment. Leave this new class as it is for now, and instead click on the class tab again *CourseStatistics*. Click again on the incorrectly marked class *Course* and create this class in the same way as you did the class *Student* before.

Then go back to the *course statistics* class again. Eclipse now presents this class slightly differently than before. What do you notice about the new presentation (Figure 20.12)? Noticeably, the underlines have disappeared for the *Student* and *Course* classes. Eclipse recognizes the *Student* and *Course* data types because you created the corresponding classes. However, the Eclipse IDE marks the constructors as incorrect. This is because the classes are still empty and the constructors are missing.

To change this, click on underlined constructor of class *course* inside class *CourseStatistics* and select from context menu that appears CREATE CONSTRUCTOR 'COURSE(INT, STRING)' (Figure 20.13). Proceed with the *Student* class in the same way and choose the CREATE CONSTRUCTOR 'STUDENT(STRING, STRING, STRING, COURSE)' command.

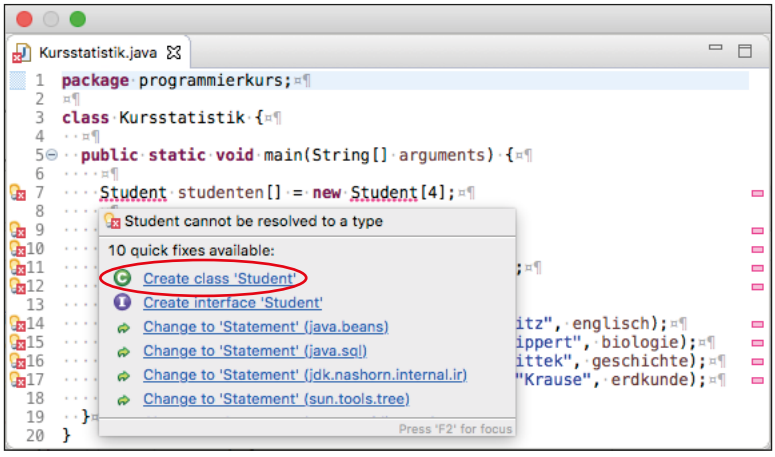


Figure 20.11: Using the programming help to create the new class »Student«.

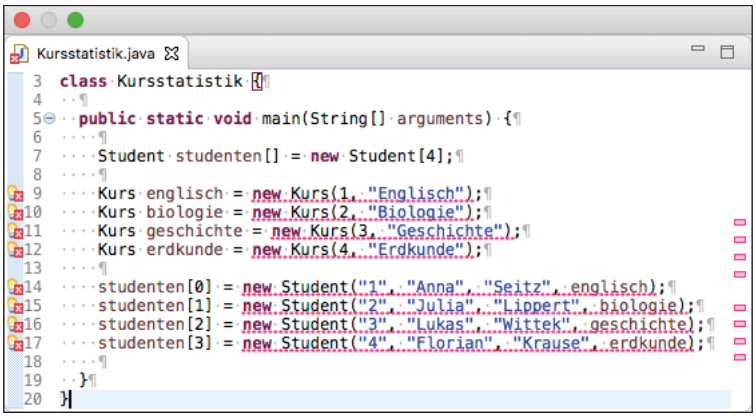


Figure 20.12: Eclipse flags constructors as broken.

After creating the constructors, Eclipse should render the *course statistics* class without errors. In the next step, you need to fully implement the two classes *student* and *course*. Don't use Eclipse wizards this time, but correct the class by hand using the Eclipse"=editor according to the following template:

Listing 20.2: The attributes of the class »Student«.

```
1  package programmierkurs;
2
3  public class Student {
4
5    String matrikelnummer;
6    String vorname;
7    String nachname;
8    Kurs wahlpflichtfach;
```

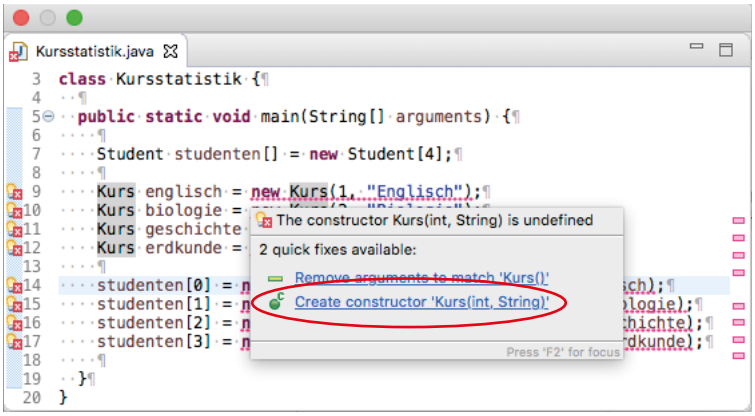


Figure 20.13: Using the programming help to create the constructors.

```
9
10   public Student(String matrikelnummer, String vorname,
11                  String nachname, Kurs wahlpflichtfach) {
12     super();
13     this.matrikelnummer = matrikelnummer;
14     this.vorname = vorname;
15     this.nachname = nachname;
16     this.wahlpflichtfach = wahlpflichtfach;
17   }
18
19   public String getMatrikelnummer() {
20     return matrikelnummer;
21   }
22
23   public void setMatrikelnummer(String matrikelnummer) {
24     this.matrikelnummer = matrikelnummer;
25   }
26
27   public String getVorname() {
28     return vorname;
29   }
30
31   public void setVorname(String vorname) {
32     this.vorname = vorname;
33   }
34
35   public String getNachname() {
36     return nachname;
37   }
38
39   public void setNachname(String nachname) {
40     this.nachname = nachname;
41   }
42
```

```
43    public Kurs getWahlpflichtfach() {
44      return wahlpflichtfach;
45    }
46
47    public void setWahlpflichtfach(Kurs wahlpflichtfach) {
48      this.wahlpflichtfach = wahlpflichtfach;
49    }
50  }
```

After that, go to the marked class *course* and complete these classes also using the Eclipse editor according to the following template:

Listing 20.3: The class »Course«.

```
1  package programmierkurs;
2
3  public class Kurs {
4    String id;
5    String name;
6
7    public Kurs(String id, String name) {
8      super();
9      this.id = id;
10     this.name = name;
11   }
12
13   public String getId() {
14     return id;
15   }
16
17   public void setId(String id) {
18     this.id = id;
19   }
20
21   public String getName() {
22     return name;
23   }
24
25   public void setName(String name) {
26     this.name = name;
27   }
28 }
```

What is missing now is the output of the course with the subjects taken in the main program. It should get a program output like in listing 20.4. Add the class *CourseStatistics* accordingly. The output generates a table. The string \t causes a tab feed and ensures that columnar output is produced.

Listing 20.4: The complete class »CourseStatistics«.

```
1  package programmierkurs;
2
```

```
3  /**
4   * Klasse "Kursstatistik"
5   * @author Programmierkurs Prof. Roth
6   *
7   */
8  class Kursstatistik {
9
10   public static void main(String[] arguments) {
11
12     Student studenten[] = new Student[4];
13
14     Kurs englisch = new Kurs("01-SS/2020", "Englisch");
15     Kurs biologie = new Kurs("02-SS/2020", "Biologie");
16     Kurs geschichte = new Kurs("03-SS/2020", "Geschichte");
17     Kurs erdkunde = new Kurs("04-SS/2020", "Erdkunde");
18
19     studenten[0] = new Student("100010", "Anna", "Seitz", englisch);
20     studenten[1] = new Student("100011", "Julia", "Lippert", biologie);
21     studenten[2] = new Student("100012", "Lukas", "Wittek", geschichte);
22     studenten[3] = new Student("100013", "Florian", "Krause", erdkunde);
23
24     System.out.println(
25       "Vorname\tName\tMatrikelnummer\tWahlpflichtfach\tKursnummer");
26     System.out.println(
27       "============================================================");
28     for (int i = 0; i < studenten.length; i++) {
29       System.out.println(studenten[i].getVorname() + "\t" +
30         studenten[i].getNachname()  + "\t" +
31         studenten[i].getMatrikelnummer()  + "\t\t" +
32         studenten[i].getWahlpflichtfach().getName() + "\t" +
33         studenten[i].getWahlpflichtfach().getId());
34     }
35   }
36 }
```

Compile

In the next step, Eclipse translates the file *CourseStatistics.java* using Eclipse's Java compiler. Instead of »translating«, you also say »compiling«. Compiling an entire project is called a »build«. During the build process, so-called *bytecode* is created from the source code. Bytecode is a special form of binary code for the virtual machine. It is not machine code for a specific processor and does not contain calls to operating system functions.

Bytecode is an intermediate format, which chapter 21, »Runtime Environment«, section 21.3, »Bytecode«, discusses in detail. At this point, the only important thing is that the compiler creates a file of the same title, but with the extension *class*, for each class it translates. Thus, *CourseStatistics.java* becomes *CourseStatistics.class* (Figure 20.14).

Many development environments require you to manually trigger the build process using a menu command or key combination. By default in Eclipse, the development environment always compiles the entire project automatically in the background. You set this behavior

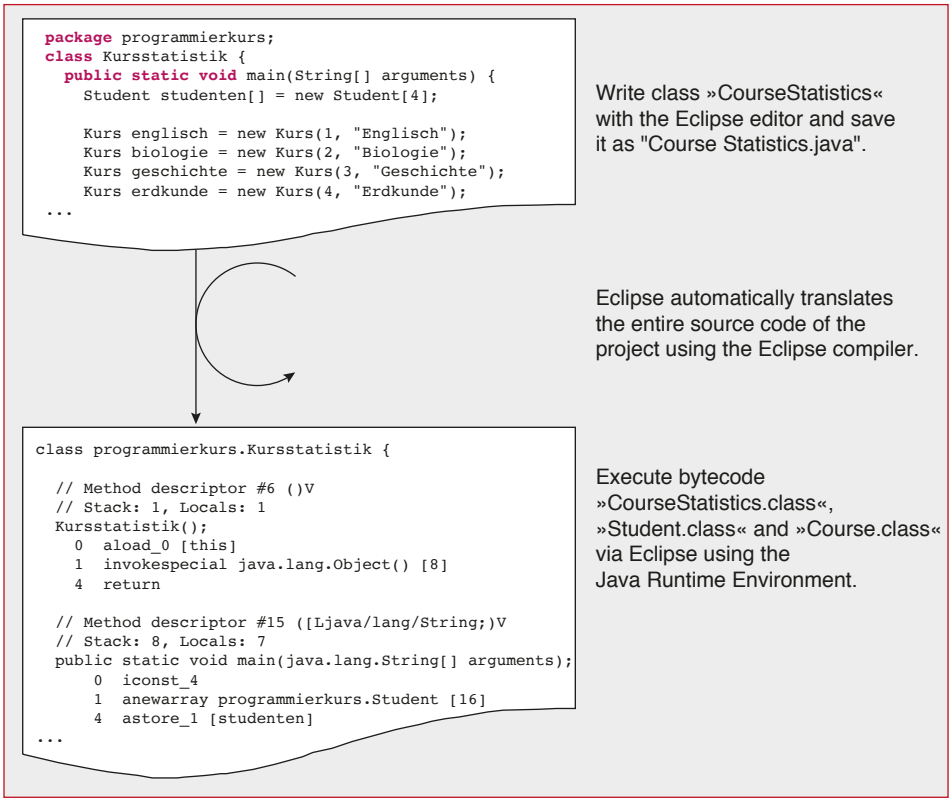


Figure 20.14: The Eclipse compiler translates the source code to bytecode.

in the Eclipse"=menu PROJECT, where a check mark is placed at the option BUILD AUTOMATICALLY if you want Eclipse to compile the project automatically. What are the advantages of this function? It has the advantage that you can run the project immediately if the project is saved and error free.

Run

Now right-click on the class *CourseStatistics* and run the program using the command RUN AS → JAVA APPLICATION. It should output the following:

```
First Name Last Name Matriculation Number Elective Course Number
============================================================
null Seitz 100010 English 01 - SS/2020
null Lippert 100011 Biology 02 - SS/2020
zero Wittek 100012 History 03 - SS/2020
null Krause 100013 Geography 04 - SS/2020
```

Everything looks fine except for the first names. What is the reason for the output of the value *null*?

Debugging

To track down the error, you use a debugger. The JDK contains a very simple debugger. Much more powerful is the Eclipse Debugger, with which you can search for errors very efficiently. Since something seems to be wrong with the output of the *getFirstName()* method, it makes sense to call the *Student* class where this method is defined. Set a breakpoint in this class by double-clicking on the left margin. A blue dot should then appear (Figure 20.15).

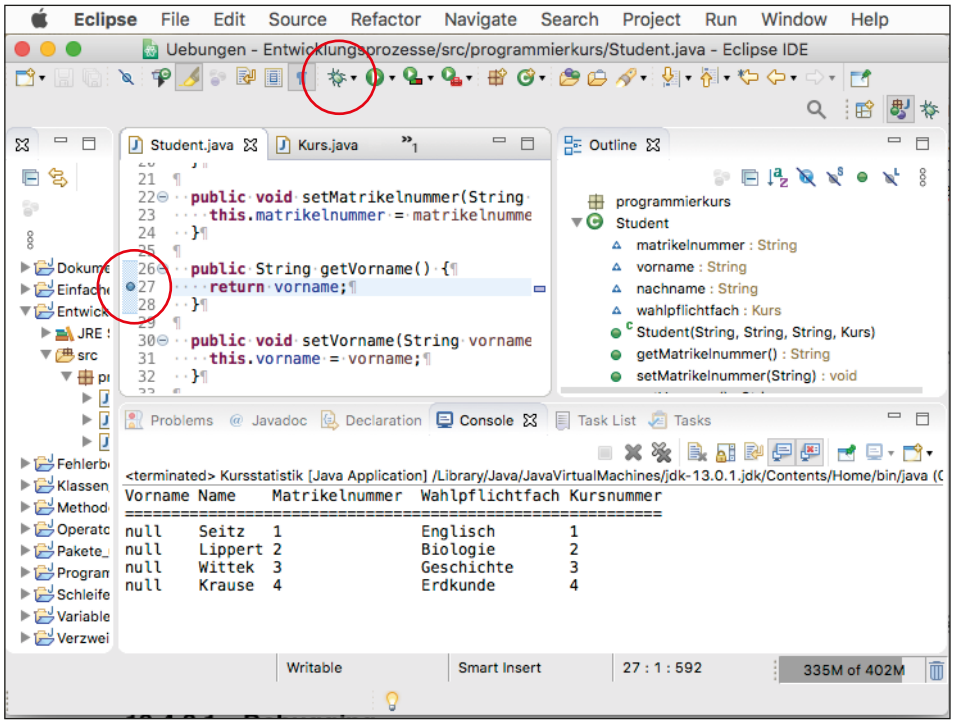


Figure 20.15: A breakpoint will be set.

After that, start the program by clicking on the icon of a beetle in the toolbar of the Eclipse IDE. A dialog of the development environment will appear asking if you want Eclipse to switch to the debugging perspective. Acknowledge the dialog by clicking the SWITCH button. The development environment then changes completely. Eclipse has stopped running the program at the breakpoint marked in blue. In the right pane you can see a display of the variables with their current value (Figure 20.16).

Expand the plot completely so you can see the details. You should then be able to see that the variable *firstName* has the value *null*. Now stop the program by clicking on the red rectangle in the lower right corner of the CONSOLE window. This will immediately terminate the program run.

Since the *Student* class is not particularly complicated, there is not much room for error. Now scroll down the class *Student* to where the constructor is and set another breakpoint where you can see the call *super()*. Now restart the program, which will stop exactly at this

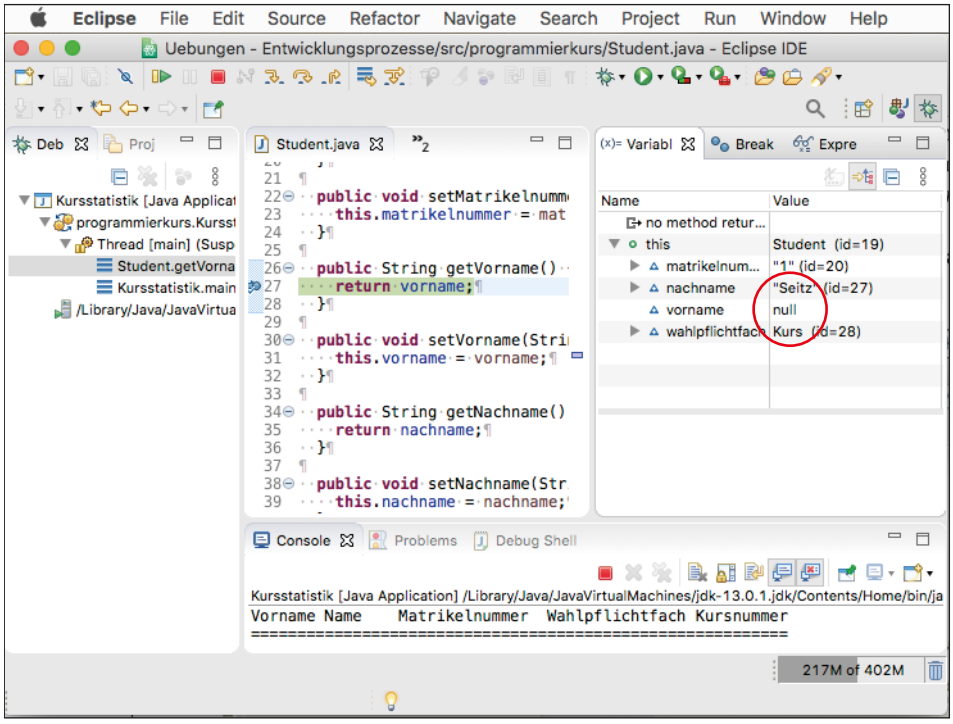


Figure 20.16: The value of variable »firstName« is »null«.

point in the constructor. On the right side you can observe the value of the variable. With the key F6 you work through instruction by instruction of the program. Debugging the constructor makes it clear that the correct value of the given name is passed. However, the constructor is missing the instruction that passes the value for the first name to the class attribute named *firstName*.

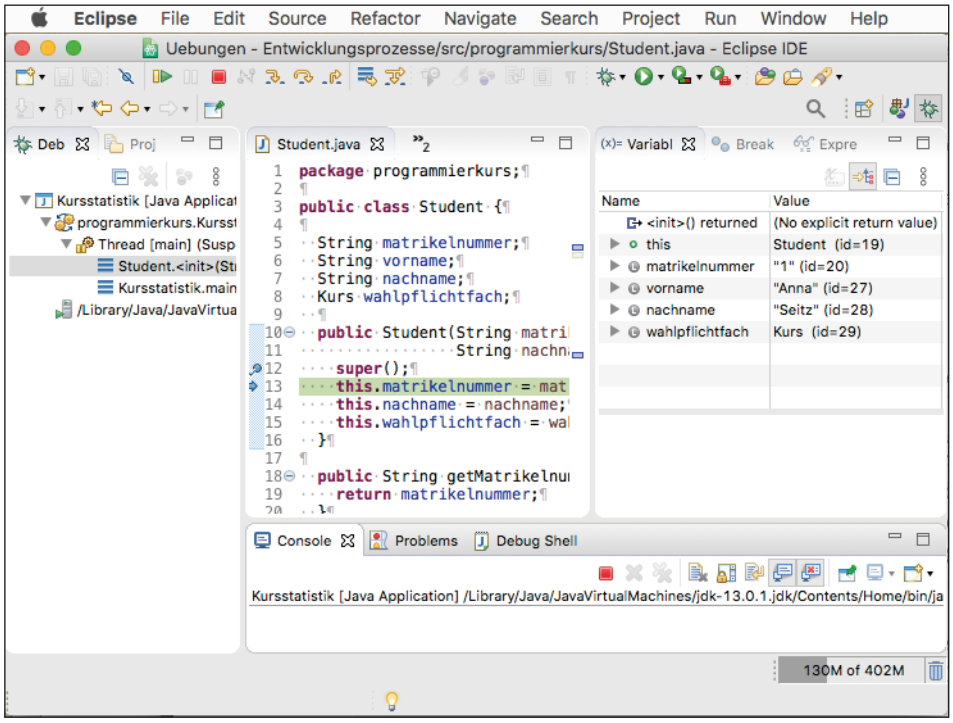


Figure 20.17: The assignment »this.firstname = firstName« is missing.

Add the following statement to the constructor during debugging:

```
this.firstName = firstName
```

You don't have to stop the program run to do this, just type in the new statement in the editor. After that, save the class. Eclipse translates the class, replaces it during the run, and restarts the program run at the breakpoint. The function to swap a class during the program run is called hot-swap debugging. You can now simply press F8 or click RESUME on the toolbar to run the program. The program will stop at any output in the getter method. You will see that the program run is successful this time, and get the following correct program output:

```
First name Last name Matriculation number Elective course number
============================================================
Anna Seitz 100010 English 01 - SS/2020
Julia Lippert 100011 Biology 02 - SS/2020
Lukas Wittek 100012 History 03 - SS/2020
Florian Krause 100013 Geography 04 - SS/2020
```

20.4.4 Test

Starting the program and checking the program output is the first simple developer test. Far more elaborate and safer is the so-called unit test. For this, a developer writes a test class for each of his classes based on the class library *JUnit*. Eclipse supports unit tests with a whole range of functions. To better understand the effect of the test, delete again the following statement from the constructor:

```
this.firstName = firstName;
```

To add test cases to your project afterwards, right-click on the *Student* class in the PACKAGE EXPLORER. Then select NEW → JUNIT TEST CASE from the context menu. Eclipse then launches a dialog to create a new JUnit test (Figure 20.18).

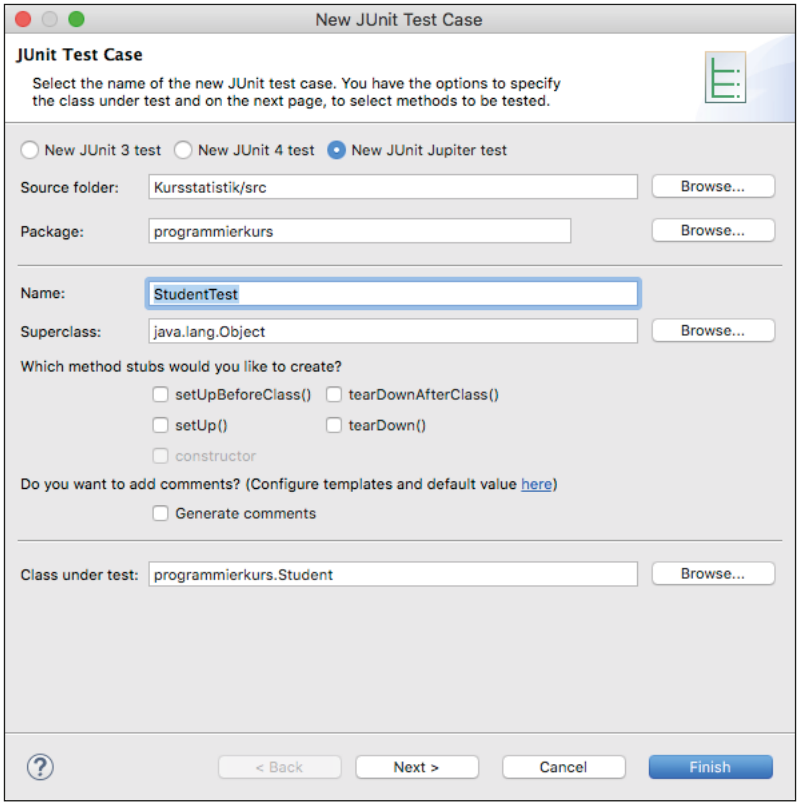


Figure 20.18: Use this dialog to create a new JUnit test case

Leave all settings as assigned by Eclipse and then click NEXT. On the second page of the dialog, select the methods for which you want Eclipse to generate test cases. Select the methods *getMatriculationNumber()*, *getFirstName()*, *getLastName()* and *getElectiveSubject()* (Figure 20.19).

After that, end the dialog by clicking the *finish* button. Eclipse then displays a dialog asking whether the library *JUnit* should be added to the »Build Path«. Close the dialog by clicking

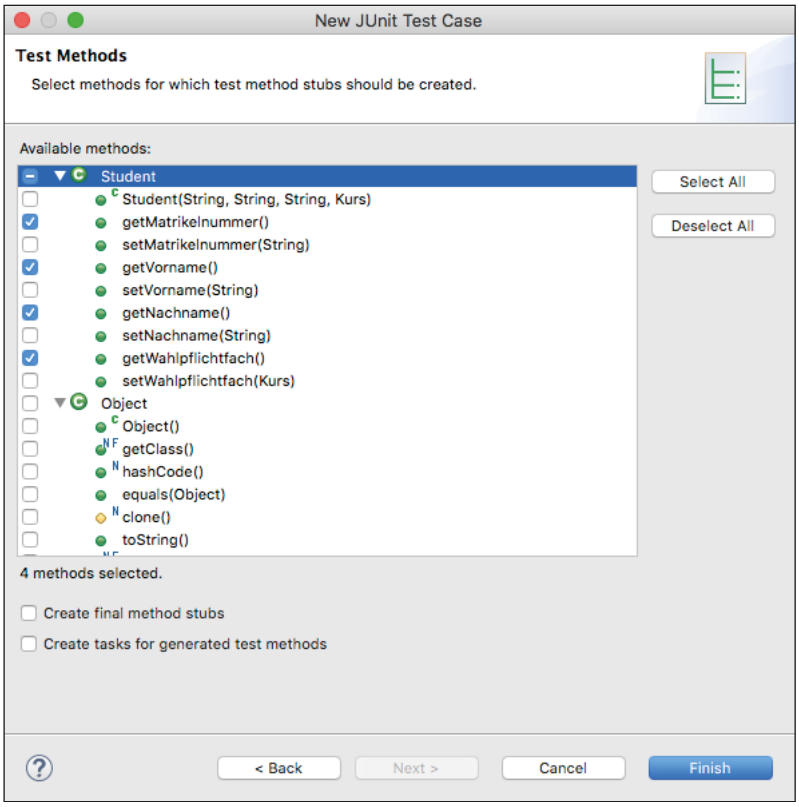


Figure 20.19: Here you select the methods for which Eclipse should generate test cases.

.

the OK button to have Eclipse add this library to the project. Then close the main dialog with FINISH. Eclipse will then create a minimal test class (20.5).

Listing 20.5: The »StudentTest« test class created by Eclipse.

```
1  package programming course;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  import org.junit.jupiter.api.Test;
6
7  class StudentTest {
8
9    @test
10   void testGetMatrikelnummer() {
11     fail("Not yet implemented");
12   }
13
14   @Test
15   void testGetFirstName() {
```

```
16      fail("Not yet implemented");
17    }
18
19    @Test
20    void testGetLastName() {
21      fail("Not yet implemented");
22    }
23
24    @Test
25    void testGetElectiveSubject() {
26      fail("Not yet implemented");
27    }
28  }
```

The class contains four method templates for implementing tests. All methods contain a call called *fail()*. This is nothing but a reminder for the developer to implement the method. Save the class and click the RUN command on the RUN menu to have Eclipse run this test. Eclipse then opens another window next to PACKAGE EXPLORER with the test results (Figure 20.20).

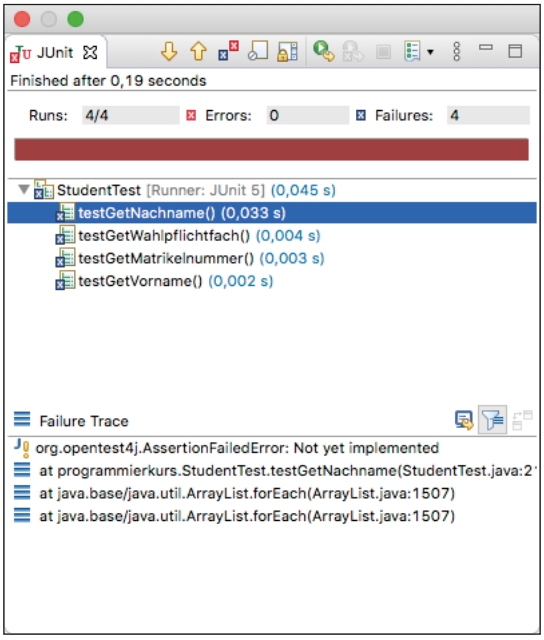


Figure 20.20: The test results after the first start.

According to Eclipse, the test of the four methods failed. This is no wonder because nothing has been implemented so far. Now modify the *StudentTest* class as follows, save the class and run the test again:

Listing 20.6: The complete class »StudentTest«.

```
1  package programmierkurs;
```

```
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  import org.junit.jupiter.api.Test;
6
7  class StudentTest {
8
9    private String kursId = "01-SS2020";
10   private String kursname = "Englisch";
11   private Kurs kurs = new Kurs(kursId, kursname);
12   private String matrikelnummer = "1000010";
13   private String vorname = "Lukas";
14   private String nachname = "Wittek";
15   private Student student =
16       new Student(matrikelnummer, vorname, nachname, kurs);
17
18
19   @Test
20   final void testGetMatrikelnummer() {
21     assertEquals(matrikelnummer, student.getMatrikelnummer());
22   }
23
24   @Test
25   final void testGetVorname() {
26     assertEquals(vorname, student.getVorname());
27   }
28
29   @Test
30   final void testGetNachname() {
31     assertEquals(nachname, student.getNachname());
32   }
33
34   @Test
35   final void testGetWahlpflichtfach() {
36     assertEquals(kurs, student.getWahlpflichtfach());
37   }
38 }
```

This time, Eclipse should indicate that there is an error in the *getFirstName()* method. This is basically the same result we already got from the normal program test. The example should show how to write unit tests using Eclipse for more complicated programs. This type of test is very important so that you know if your implementation meets the requirements.

20.4.4.1 Documentation

After the bugs are fixed, the team needs to document all the classes and their methods. This is done in the way described in chapter 18, »Documentation«. I will not go into it again here.

20.4.4.2 Integration

The next step is for the team to store all classes centrally, version them, and build a version from them. In a larger project, more classes would be built in parallel with *course statistics*, *student*, and *course*. At some point, the team assembles a version of the software from the classes that have been created at the different workstations. Because all classes are "integrated" here, this process is called integration.

After each integration, the program *course statistics* gets a new, consecutive version number. For example, you can start with version 0.1, then increase to 0.2 until you reach a version 1.0 at the end. For the integration, one team member copies the classes to his integration machine. In this case, Florian does this task on his machine (Figure 20.21).

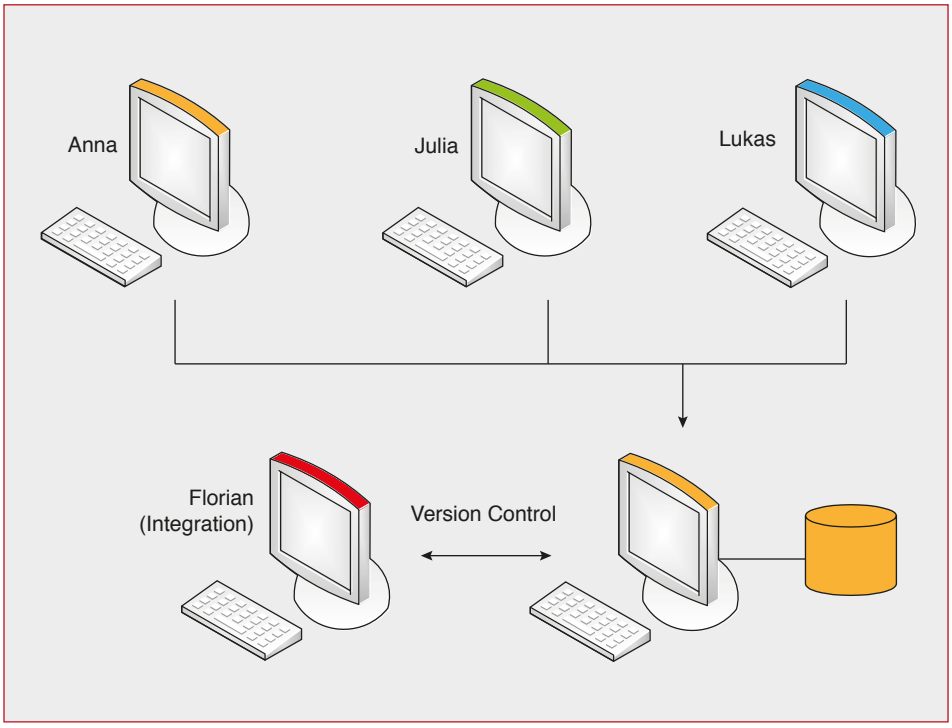


Figure 20.21: On the integration environment, building a version of the program takes place.

The integrated version must be subjected to quality assurance again using certain tools (integration test). Only when this test is successful can the team package this version of the program *course statistics* to make it available to the university as »Release Candidate No. 1«. Release Candidate No. 1 is the version to present to the customer. Maybe the customer still has one or two suggestions for improvement. Then you change that and build a »Release Candidate No. 2«. In practice, three release candidates should be enough to deliver the software.

20.5 Operating Phase

20.5.1 Deployment

It would be very impractical to deliver the bytecode to the customer in the form of many individual files. Instead, one either packages the program in a Java archive (JAR) or converted it into a native program. The customer can run both without a development environment. A tool that allows you to package classes into Java archives is also included in the JDK and is called *JAR*. Eclipse provides you with a much more convenient tool. You start this Eclipse tool from the *File* menu by using the *Export* command (Figure 20.22).

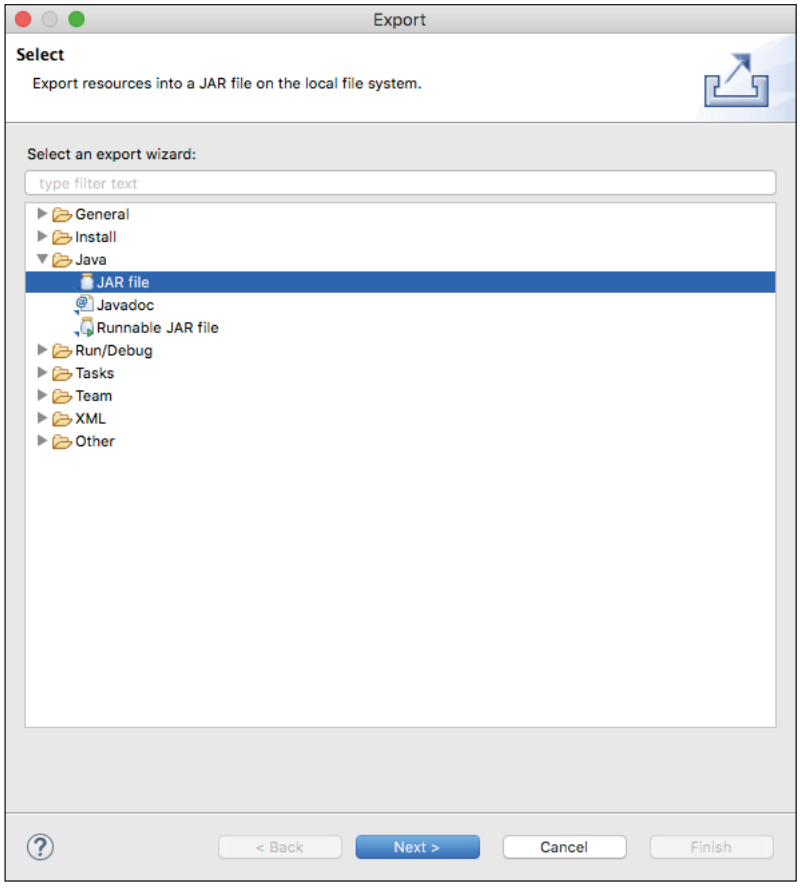


Figure 20.22: Use this dialog to export the program »Course Statistics«.

In the dialog that appears, expand the JAVA subitem, select JAR FILE, and then click the button NEXT >. The second page of the dialog displays all the projects in the *Exercises* workspace. Click on the *Development_Processes* project in the left pane of the dialog under SELECT THE RESOURCES TO EXPORT and expand the subtree of this project. From the project

resources, select only the *src* folder. This will cause Eclipse to export only the class files that belong to this source code (Figure 20.23).

Nothing should be selected on the right side of the dialog. Below the project tree you can select several options. Here EXPORT GENERATED CLASS FILES AND RESOURCES should already be selected. Under SELECT THE EXPORT DESTINATION you select the directory and the name of the JAR file. Below that you can specify if the archive should be compressed. This is recommended in any case to save disk space.

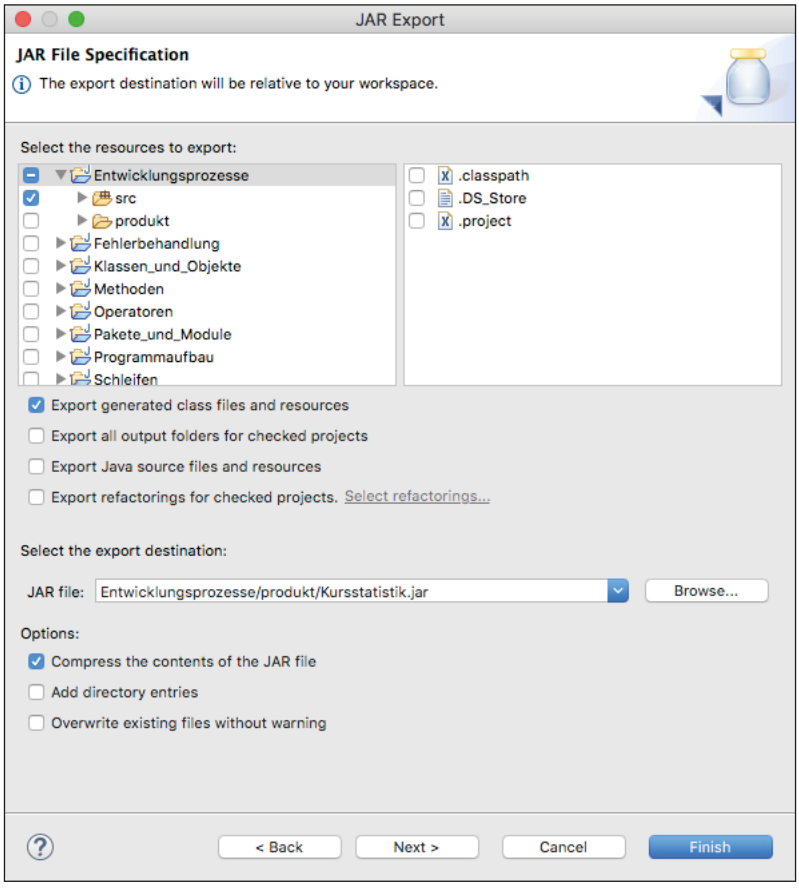


Figure 20.23: On this page you specify what should be exported.

After that, click NEXT > to go to the next page. On this page you specify how to handle compiler errors and warnings when compiling the Java archive. It is actually not common to ship class that contain compiler errors, so you can turn off the first option. Turning off the second option is risky, because many projects have warnings. If you turn off this option, the archive may remain incomplete if there are warnings in the project. After making your selection, click the NEXT > button to go to the next page.

On the last page, Eclipse compiles the so-called manifest. It defines how the program is started. You can leave the pre-selection as it is. In the lower area you can seal the archive.

The background of this function is that when you deliver professional programs, you usually encrypt the bytecode. This makes it difficult for the competition to decompile the source code of the program. For our program the function is irrelevant. More important is the last field at the bottom: Specify here *CourseStatistics* as the main class and then generate the JAR file by clicking on FINISH.

If everything worked, you will find a file named *CourseStatistics.jar* in the directory you selected on the second page of the dialog. The file is a ZIP file containing Java classes in the form of bytecode. Using such a Java archive, a Java program can be started from the Windows" command prompt or a Unix terminal using the following command:

```
java --enable-preview -jar course-statistics.jar
```

For this, it is necessary that you have installed Java correctly as described in chapter 4, »Development Environment«. The *–enable-preview* option is necessary with older Java versions because Eclipse compiled the program using the preview functions of Java 13. With Java 14, it should no longer be necessary to set this option when running the program.

If the program launch was successful, you will see the course statistics in the terminal just as they were previously seen within the development environment (Figure 20.24). If this is the case, the development as well as distribution of the program is finished. This brings us to the last phase in the life cycle of a program - operation.

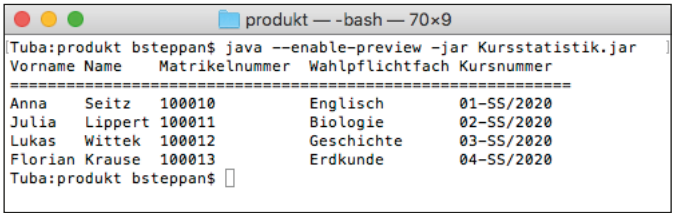


Figure 20.24: The output of the program at startup from the terminal.

20.5.2 Maintenance

Once the project is finished and the program *CourseStatistics* is installed in its final form in the university, the part of the operational phase called maintenance begins. In practice, this usually means that small errors are reprimanded by the customer or the customer wants some improvements made. In the case of software bugs, our team would have to modify, test, package and re-deliver the program *CourseStatistics* program. In the case of improvement requests, the university would in turn have to consider whether to place a follow-up order with our team.

20.6 Summary

The software development process proceeds in three main phases: planning, construction, and operation. The planning phase is dominated by requirements gathering. The con-

struction phase consists of analysis and design, implementation, and testing of the software, while operation is dominated by distribution and maintenance.

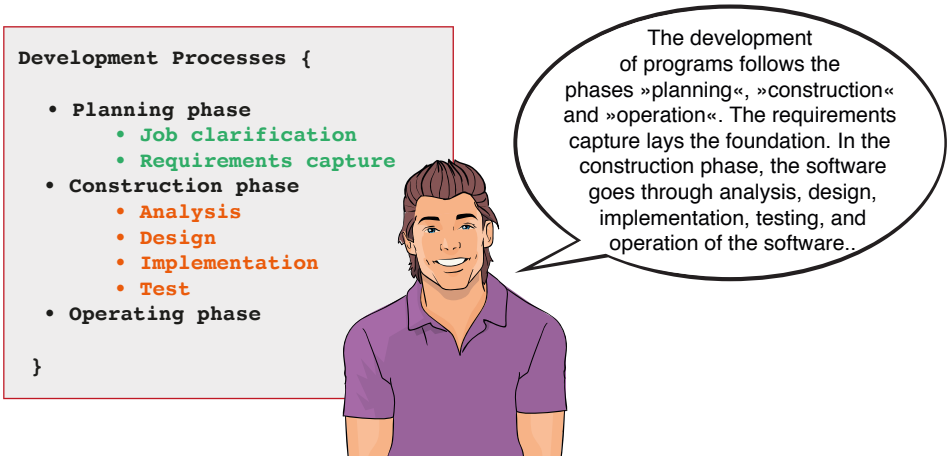


Figure 20.25: The development of a program goes through three phases.

20.7 Literature

Bernhard Steppan: »Verpackungskünstler«; Codewrapper für Java Applications;
https://shop.heise.de/katalog/verpackungskunstler-2

20.8 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/development_processes
There you will also find the solutions to the exercises.

21 Runtime Environment

21.1 Introduction

The Java runtime environment is the core component of the Java platform. Together with the bytecode, it ensures the high portability of Java programs. This chapter focuses on the virtual machine, the so-called bytecode and the Java compiler. You will learn how the virtual machine is structured, how bytecode is created and why Java programs can be transferred so easily from one operating system to another.

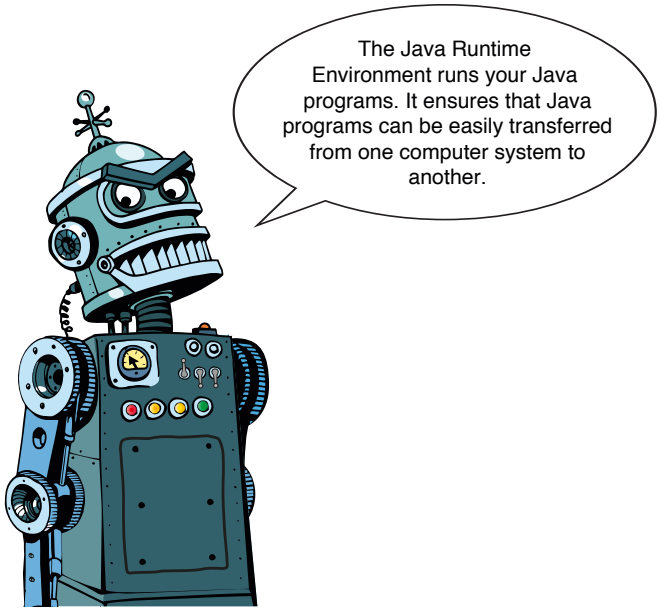


Figure 21.1: A Java runtime environment also works inside Robert.

21.2 Overview

You cannot possibly hand over a Java program that you have developed to the end user in the form in which it exists in the development environment. After all, the end user expects to be able to run your program without Eclipse. He expects your program to behave no differently than other programs on his operating system.

In the last chapter, you saw how to deliver a Java program. To do this, you used Eclipse to package the Java program as a Java archive. This Java program could then be easily started from a terminal of the operating system with the command *java -jar Kursstatistik.jar*. This was still not optimal, but at least it was better than starting with a development environment.

How did this work? To better understand the process, let's briefly return to the starting point of this book in chapter 2, »Technology Overview«. The Java installation on your computer is called the Java Development Kit (JDK). As the name of this software package suggests, the installation consists of various software development tools. Included is the Java Runtime Environment (Figure 21.2).

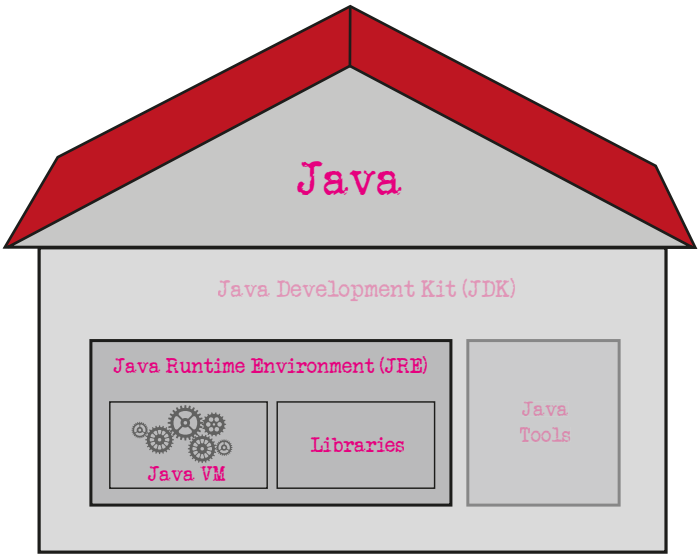


Figure 21.2: The Java runtime environment consists of libraries and the Java VM.

We have given the Java Development Kit tools the cold shoulder so far. The reason for this disinterest was that Eclipse, as a universal tool, replaces all JDK tools within its integrated development environment. Compiler, debugger, editor - for everything Eclipse has a more than adequate replacement. So why install the Java Development Kit? What Eclipse needs from this Java installation is the Java Runtime Environment (JRE). Without this Java Runtime Environment, Eclipse would not be able to start, since it is a Java program itself. In addition, Eclipse would not be able to run your Java programs either.

This Java runtime environment mainly consists of the so-called Java Virtual Machine (JVM) and various libraries. You have used the libraries a lot so far, because they contain, among

other things, classes like the class *String* for displaying strings or the class *System* for program output. What remains to be clarified is the virtual machine and the bytecode.

21.3 Bytecode

In order to understand why the Java inventors developed a virtual machine, it is necessary to learn a bit about bytecode. To do this, start Eclipse with the workspace *exercises*. Then copy the project *Development Processes* of the previous chapter and name the new project after this chapter *Runtime Environment*. After that, close all files in the editor of Eclipse again.

Now locate the new project directory on your computer's hard drive. You can find the directory relatively easily using Eclipse. To do this, right-click on the project name in Eclipse's PACKAGE EXPLORER. From the menu that Eclipse displays, select the PROPERTIES command. This brings up a dialog with the project properties (Figure 21.3).

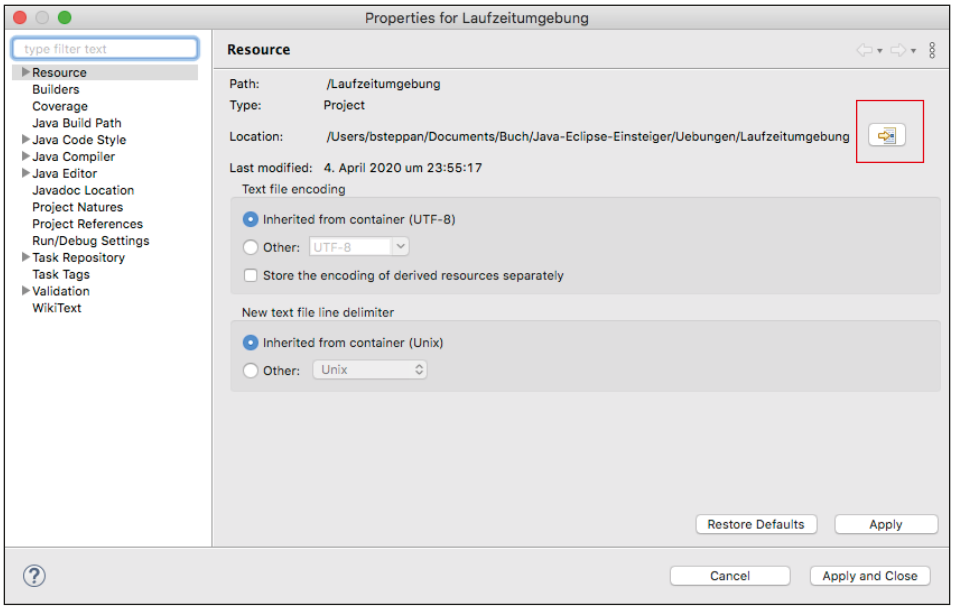


Figure 21.3: Project properties also shows the location on disk.

In this dialog, the RESOURCE item of the tree structure is selected on the left side. On the right side, the LOCATION item lists the path where Eclipse saved the project. If you click on the icon at the end of the path listing, Eclipse opens an OS view that shows the structure of the project directory. It should resemble the following figure 21.4, except for the directory named *product*.

At the top is the project directory named *runtime environment*. After that comes the *bin* directory, which contains the package *programmingcourse*. This package contains the bytecode of the four classes of the program. The directory with the source code named *src* is

built in a mirror image. Also in this directory is the same package with the four classes in source code.

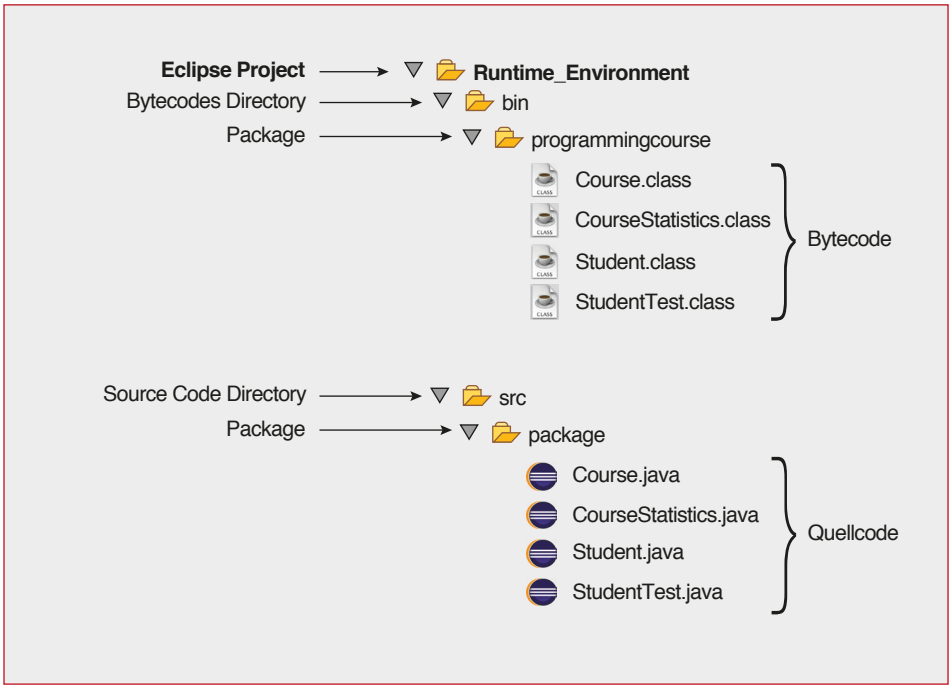


Figure 21.4: Source code and the bytecode of the project.

Now click on the Eclipse menu FILE and select OPEN FILE from the menu. Then navigate to the *bin* directory of the *runtime environment* project and open the *course.class* file. Eclipse will then display the binary code of this class in the editor (Listing 21.1).

Listing 21.1: Excerpt from the bytecode of the class »Course«.

```
1  / Compiled from Kurs.java (version 13 : 57.65535, super bit)
2  public class programmierkurs.Kurs {
3
4    // Field descriptor #6 Ljava/lang/String;
5    java.lang.String id;
6
7    // Field descriptor #6 Ljava/lang/String;
8    java.lang.String name;
9
10   // Method descriptor #9 (Ljava/lang/string;Ljava/lang/string;)V
11   // stack: 2, locals: 3
12   public course(java.lang.String id, java.lang.String name);
13      0 aload_0 [this]
14      1 invokespecial java.lang.Object() [11]
15      4 aload_0 [this]
16      5 aload_1 [id]
```

```
17       6 putfield programming.course.id : java.lang.String [14]
18       9 aload_0 [this]
19      10 aload_2 [name]
20      11 putfield programming_course.name : java.lang.String [16]
21      14 return
22        Line numbers:
23          [pc: 0, line: 8]
24          [pc: 4, line: 9]
25          [pc: 9, line: 10]
26          [pc: 14, line: 11]
27        Local variable table:
28          [pc: 0, pc: 15] local: this index: 0 type: programmingcourse.
                course
29          [pc: 0, pc: 15] local: id index: 1 type: java.lang.String
30          [pc: 0, pc: 15] local: name index: 2 type: java.lang.String
31
32  (...)
33
34    // Method descriptor #23 ()Ljava/lang/String;
35    // stack: 1, locals: 1
36    public java.lang.String getName();
37      0 aload_0 [this]
38      1 getfield programming.course.name : java.lang.String [16]
39      4 areturn
40        Line numbers:
41          [pc: 0, line: 22]
42        Local variable table:
43          [pc: 0, pc: 5] local: this index: 0 type: programmierkurs.Kurs
44
45    // Method descriptor #25 (Ljava/lang/String;)V
46    // stack: 2, locals: 2
47    public void setName(java.lang.String name);
48      0 aload_0 [this]
49      1 aload_1 [name]
50      2 putfield programming.course.name : java.lang.String [16]
51      5 return
52        Line numbers:
53          [pc: 0, line: 26]
54          [pc: 5, line: 27]
55        Local variable table:
56          [pc: 0, pc: 6] local: this index: 0 type: programmierkurs.Kurs
57          [pc: 0, pc: 6] local: name index: 1 type: java.lang.String
58  }
```

Byte code consists of a sequence of bytes - hence its name. Each byte corresponds to an instruction (opcode) for the virtual machine. The virtual machine, unlike a physical machine, does not have registers. Instead, it places all variables on a stack, the operand stack. When the program is executed, the stack serves as a short-term memory for numerical values that the virtual machine is currently juggling.

How the bytecode is structured and how the virtual machine executes it becomes clear when you contrast source code and bytecode (Figure 21.5). In the upper part you can see

the *setName()* method as the source code of the *course* class, and in the lower part the bytecode generated by the Java compiler. The digits in front of the bytecode are the addresses of the instructions.

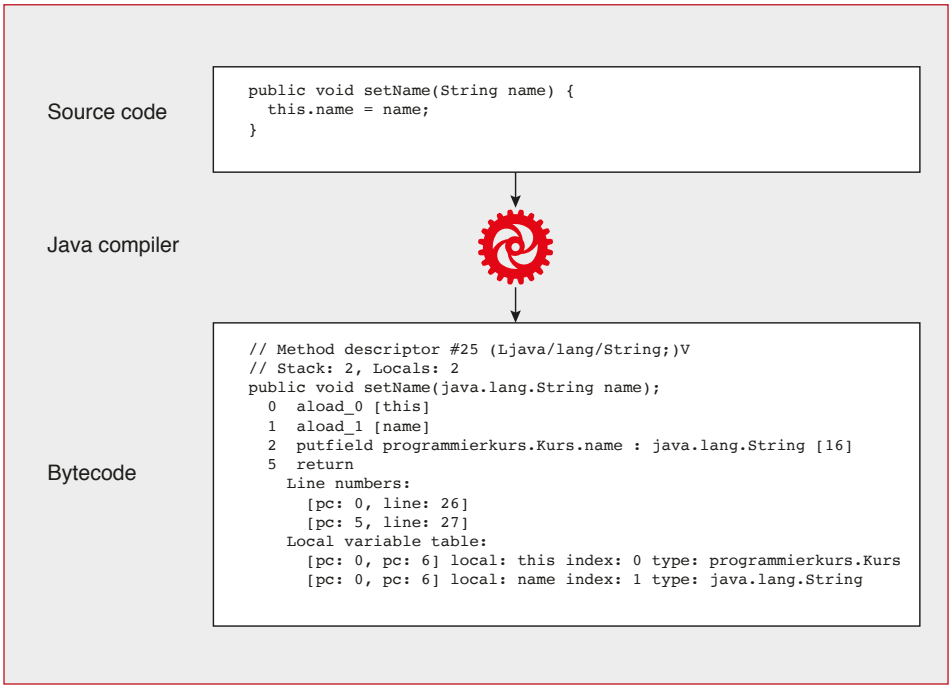


Figure 21.5: the compiler translates source code to bytecode.

The first instruction of the bytecode *aload_0* causes the virtual machine to place the value for *this* on the operand stack. The subsequent instruction results in the value of the local variable *name* also being placed on this stack. At the end follows the assignment of the value of the local variable *name* to the attribute *this.name*.

This should also be the end of the excursion into the world of bytecode. Fortunately, as a Java programmer, you don't have to deal with bytecode. However, you should understand that bytecode is nothing more than an intermediate format. This intermediate code is located somewhere between the Java source code and the machine code of a real computer. This intermediate code is executed by the virtual machine.

21.4 Java Virtual Machine

21.4.1 Artificial Computer

The heart of the Java platform is the so-called Java Virtual Machine (JVM) or simply virtual machine. Behind this term is a computer program that exists for all sorts of operating

systems and computer hardware as part of the Java installation. This software mimics a physical computer. Since it only imitates such a machine, this software is called a »virtual machine«. This virtual machine is a fairly complex piece of software (Figure 21.6).

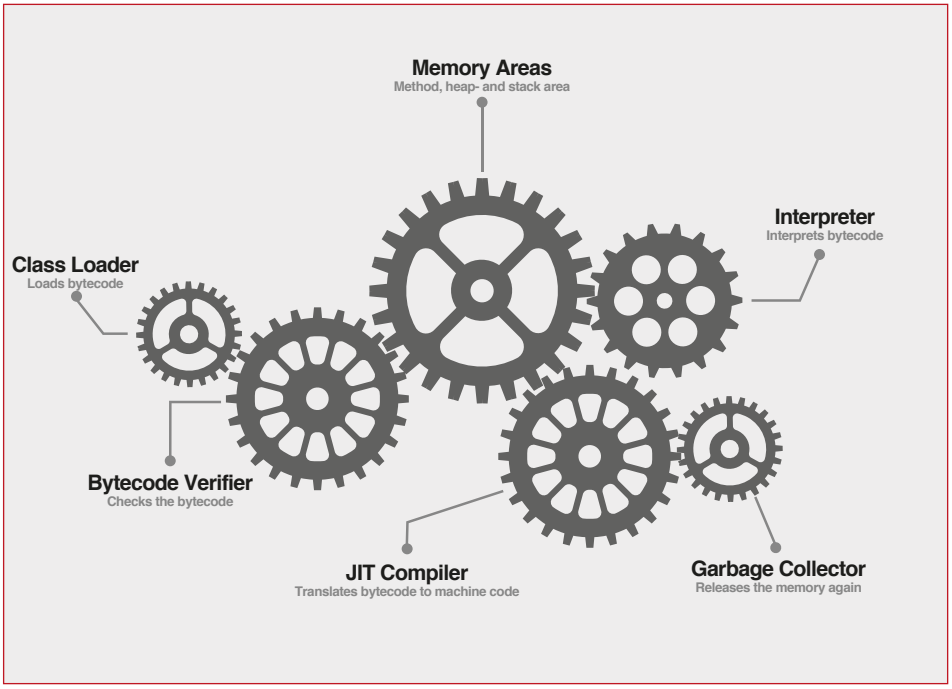


Figure 21.6: The very simplified illustrated structure of a modern Hotspot VM.

A modern Java VM consists of (simplified representation of) the class loader, the bytecode verifier, the memory space, a JIT compiler, the interpreter, and the garbage collector. When the program starts, the virtual machine uses the class loader to load the classes of a program into its memory space, then checks the bytecode using the bytecode verifier and, after a successful check, executes the Java program. This execution can be done in several ways: in pure interpreter mode, in pure mode with a JIT compiler, or via a mixed mode that combines both approaches.

21.4.2 Interpreter mode

The slowest way to run a Java program is in interpreter-only mode. In this mode, the virtual machine interprets each instruction of the bytecode and executes it itself. By processing each instruction of the bytecode itself, there is no conversion of the bytecode to native machine code. In newer Java virtual machines, this mode is disabled due to slow processing speed. You can turn it on by using the *java -Xint* instruction (Section 21.8, »JVM Configuration«).

21.4.3 JIT compiler mode

Up to 40 times faster than in interpreter-only mode, the virtual machine executes Java programs when it uses its just-in-time (JIT) compiler. In this mode, the JIT compiler translates byte code into real machine code step-by-step. As a result, a Java program is usually much faster compared to the interpreter mode. Only where parts of the program are run through very infrequently does the JIT technique have disadvantages in the execution speed of a Java program compared to interpreter mode.

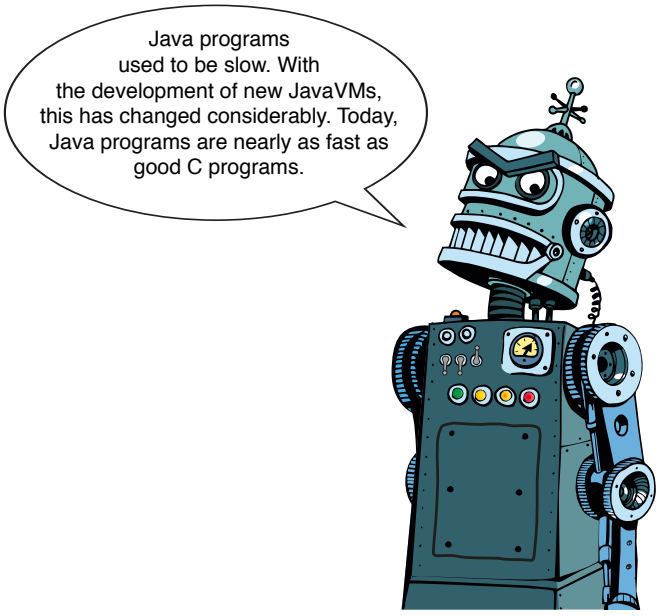


Figure 21.7: New Java VMs accelerate the execution of Java programs.

Since the speed advantages of JIT technology outweigh interpreter operation, pure JIT operation was the technique of choice until JDK 1.2 if you wanted to optimally accelerate Java programs. Meanwhile, a modern Java VM executes bytecode in a mixed mode.

21.4.4 Hotspot Mode

Between the release of Java version 1.2 and version 1.3, then-Java owner Sun Microsystems introduced a new technique for accelerating Java programs: the hotspot VM. This modern Java VM combines the advantages of interpreter technology with those of a JIT compiler.

The Hotspot VM does not translate the entire application into machine code, but only the parts where optimizations are really worthwhile. These parts are the »hot spots« of the program. The hotspot VM is still the state of the art in Java VM today.

21.4.5 Garbage Collector

The virtual machine provides automatic memory management to Java programs. Each object is automatically allocated the memory it needs by the virtual machine. The Java developers do not have to worry about the memory management of their program.

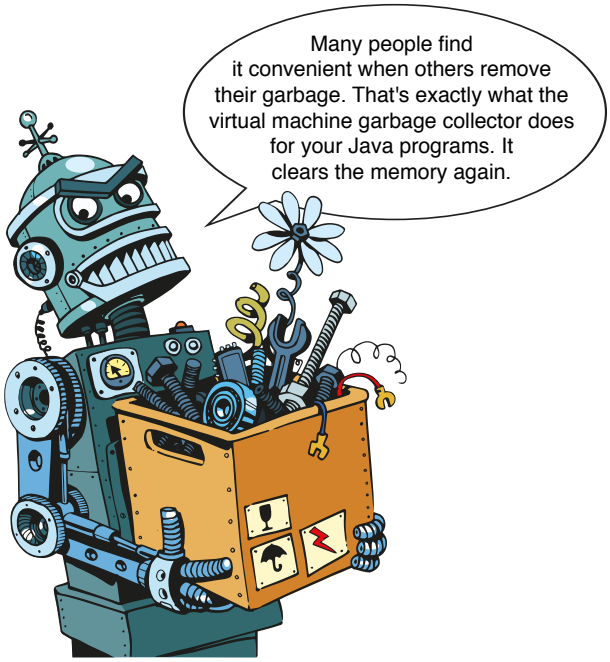


Figure 21.8: Robert gets rid of his garbage like a garbage collector.

When objects are no longer needed, the garbage collector disposes of them. At regular intervals, it checks whether it can free up memory space. Objects that are no longer needed are, so to speak, the garbage that the garbage collector »disposes« of.

21.5 Libraries

21.5.1 Native Libraries

The Java Runtime Environment consists not only of the virtual machine, but also of various native libraries that the virtual machine requires.

21.5.2 Class Libraries

In the baggage of the runtime environment there are also various class libraries. You have already used their classes several times. Chapter 22, »Class Libraries«, introduces these libraries in detail.

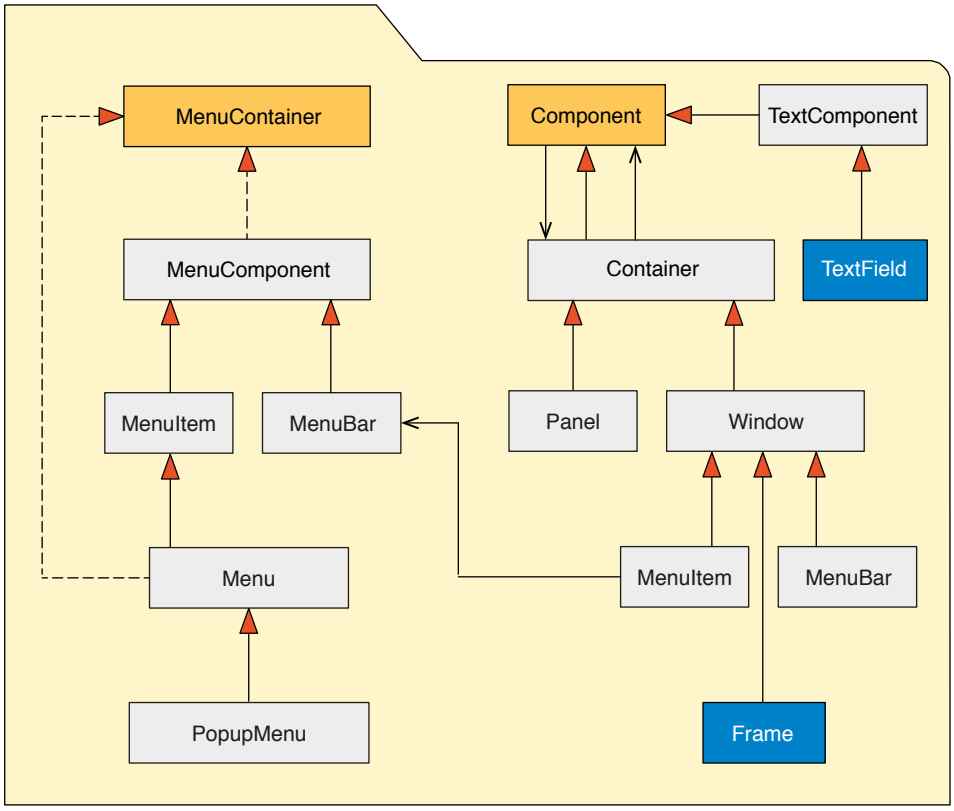


Figure 21.9: The Java VM comes with various class libraries (here: AWT).

21.5.3 Resources And Property Files

In addition, the Java VM includes mouse cursors, character sets, and a number of property files.

21.6 Portability

21.6.1 Binary Compatible Bytecode

Java programs are portable because of their bytecode and virtual machine. The same bytecode can be run from a virtual machine on Windows, Linux, and on macOS without modification. Java VMs provide a unified virtual machine for each Java program (Figure 21.10).

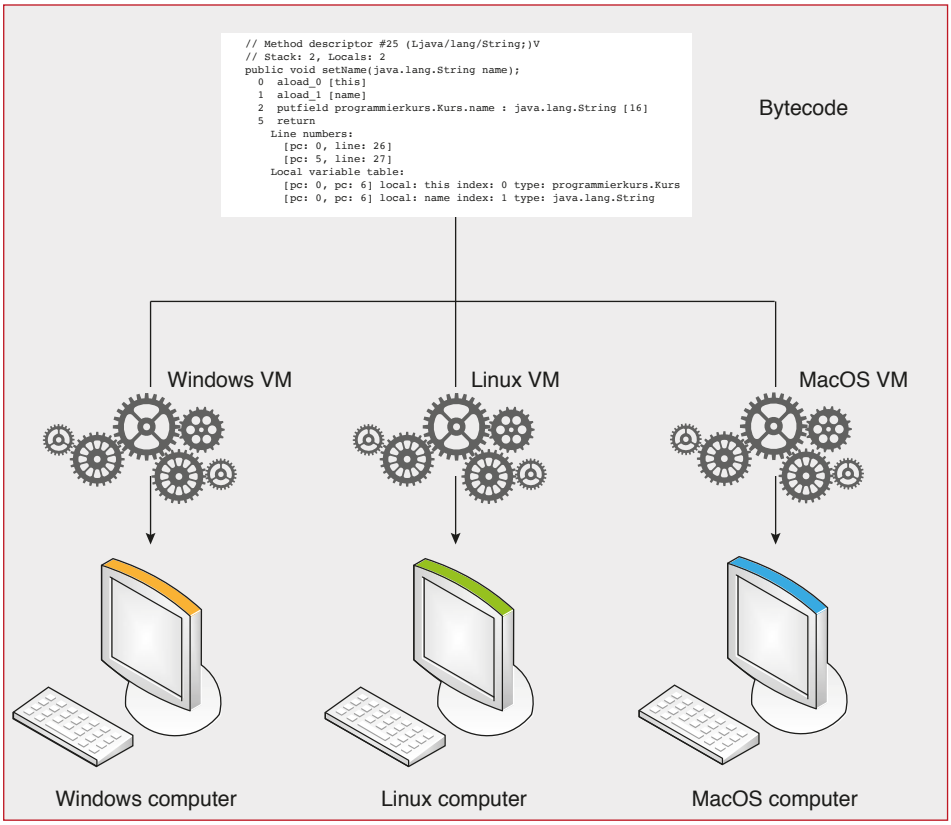


Figure 21.10: A Java program can be executed unchanged on diverse systems.

21.6.2 Porting Prerequisites

Correct Java version

Specify a minimum Java version that your program requires. If the target computer does not have this minimum version, your program should abort execution. How to check these prerequisites at startup is explained in 21.7, »Program Start«.

Correct Java Class Libraries

As memory becomes less of an issue, some professional programs ship with a Java VM. Because this JRE may be distributed, there are no additional costs. Advantage: Your program contains in any case a Java VM with the appropriate Java class libraries.

All classes

Besides a compatible virtual machine, the most important requirement for a successful port is that all classes are exported completely. This should be self-evident, but is not trivial for large programs. Development environments such as Eclipse help with the selection of the required classes and package them in JAR archives, so that this critical process is also defused.

21.7 Program Start

21.7.1 Start Script

To ensure that the end user can easily start a Java program that consists of JAR archives, you should ship your program with a startup script. This startup script can be executed by the end user again with a double click. These scripts differ in structure and file extension from operating system to operating system. Therefore it is necessary to deliver a special script for each operating system.

Windows

For Windows, a batch file is sufficient to start a Java program. It must have the extension *bat* or *cmd*. An example of a very simple script is shown in Listing 21.2.

Listing 21.2: A simple startup script for Windows.

```
1 @echo off
2 clear
3 REM "Bitte Pfad anpassen:"
4 cd /Users/bsteppan/Uebungen/Laufzeitumgebung/produkt
5 java --enable-preview -jar Kursstatistik.jar
6 @echo on
```

Unix

On Linux or macOS, you also start the program via a script. An example of a very simple shell script is shown in Listing 21.3.

Listing 21.3: A simple startup script for Unix.

```
1 #!/bin/sh
```

```
clear
# "Bitte Pfad anpassen:"
cd /Users/bsteppan/Uebungen/Laufzeitumgebung/produkt
java --enable-preview -jar Kursstatistik.jar
```

21.7.2 Native Wrapper

There are programs such as »Launch4j« or »JWrapper« that can wrap a JAR file so that the archive resembles a native program. The following example shows for Windows how the process works with Launch4j. To achieve the desired effect, you need to modify the program *CourseStatistics* a bit. To do this, open the class *Course Statistics* inside the project *Runtime_Environment* and add the following import statement for the class *Scanner* in line 3:

```
import java.util.Scanner;
```

After that, add the class *CourseStatistics* starting at line 32, as seen in Listing 21.4. By adding the statement starting at line 32, the program waits for the user to press the return key. If these instructions were missing, the program would start but end so quickly that you would not see the program output.

Listing 21.4: These applications ensure that the program does not end automatically.

```
    Scanner input = new Scanner(System.in);
    System.out.print("Schließen mit <Return> ...");
    input.nextLine();
    input.close();
```

After that, create a JAR archive again following the instructions of chapter Development Processes (section 20.5.1, »Deployment«, page 335) and save the JAR file again under the name *CourseStatistics.jar* in the *product* folder of the project. When doing this, overwrite the old version that is from the previous chapter.

Then download »Launch4j« as a zip archive from the program homepage *https://sourceforge.net/projects/launch4j*. If you unpack the archive, you will find a program called *Launch4j.exe*, which can be started without installation. The program has very many setting options, which are arranged on the tabs of the main window by categories.

On the first tab called BASIC you specify the name of the executable file and where Launch4j should save the file. Below that, you specify the JAR archive created with Eclipse and the path to it. In addition, you can specify a Windows icon that the file should have. It must be in ICO format.

In COMMAND LINE ARGS you specify parameters to be passed when the program is called. See section 21.8, »JVM Configuration«, for more information. The JAVA DOWNLOAD AND SUPPORT option can be important if you want to install your program on an unknown computer. You use this option to specify what should happen if there is no Java installation on the target computer (Figure 21.11).

You can skip the CLASSPATH, HEADER and SINGLE INSTANCE tabs. However, the JRE tab is important. There you can specify a Java"=runtime environment that you ship with your

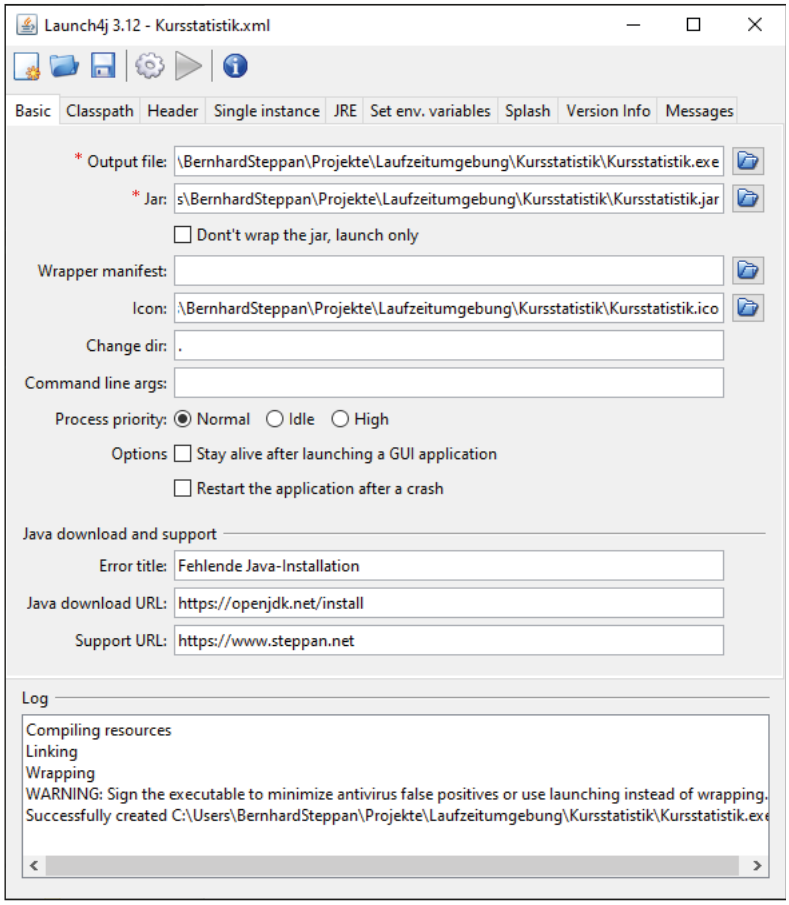


Figure 21.11: The first page of the program »Launch4j« with the basic data.

program. Below that, specify which minimum and maximum Java version your program requires. Specify version 13 here. The next options will only become clearer with the next section 21.8, "JVM Configuration".

After that, save the configuration by clicking the disk icon on the toolbar. Next, click on the gear next to the diskette icon to generate the *coursestatistics.exe* program. If everything worked, there should be a Windows program named *Course Statistics.exe* in the desired destination directory. You can start it by double clicking on it from the Windows"=Explorer.

The program opens a Windows terminal and outputs the course statistics in the same way as before with a startup script (Figure 21.12). If you click the return key afterwards, the window closes again.

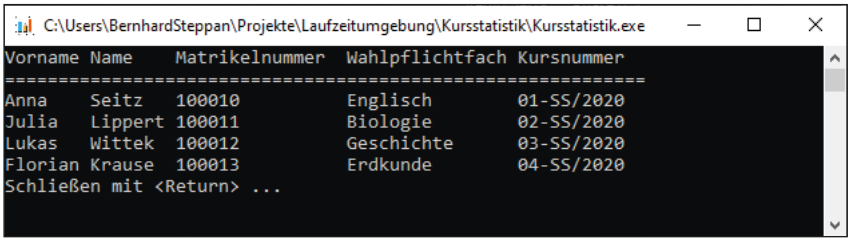


Figure 21.12: The program output of the program »Kursstatistik.exe« under Windows.

21.8 JVM Configuration

The Java VM provides many basic options for program startup. Table 21.1 shows a selection of the most important parameters. The option *-cp* means »Class Path« and is used if you want to start a program from single classes without a JAR archive. The *-version* option is also important. We used it in chapter 4, »Development Environment«, to determine the Java version using *java -version*.

Table 21.1 The most important basic options for starting a Java program

Options	Meaning
-cp <path>	Path to Java classes
-D<name>=<value>	Set system settings
-jar [-options]<filename>	Start program from jar archive
-verbose [:class|gc|jni]	Detailed information about loaded classes, garbage collector and Java Native Interface
-version	Show version and exit
-showversion	Show version and don't exit
-?, -help	Print these (and other) options
-X	Display messages that are not part of the standard

In addition to the base options, there are a number of advanced options. Table 21.2 lists the most important ones. Especially important for larger programs are the parameters *-Xms* and *-Xmx*, which determine the size of main memory for your program. You use the first value to set the initial heap size, and the second to set the maximum. Eclipse sets the first value to *-Xms256m* and the second to *-Xmx1024m*.

21.9 Summary

Java is not only an object-oriented programming language, but also a platform. This platform, which at its core consists of the virtual machine, executes programs that consist of

Table 21.2: The main advanced options for starting a Java program.

Options	Meaning
-Xint	Pure interpreter mode
-Xbootclasspath:/a >path>	Search path for initialization
-Xincgc	Enable incremental garbage collection
-Xmixed	Mixed mode (HotSpot = default) -
-Xms <size>	Initial heap size
-Xmx <size>	Maximum heap size
-Xnoclassgc	Turns off the class garbage collector

bytecode. Bytecode is a special binary format of a program for an artificial computer, a software called virtual machine.

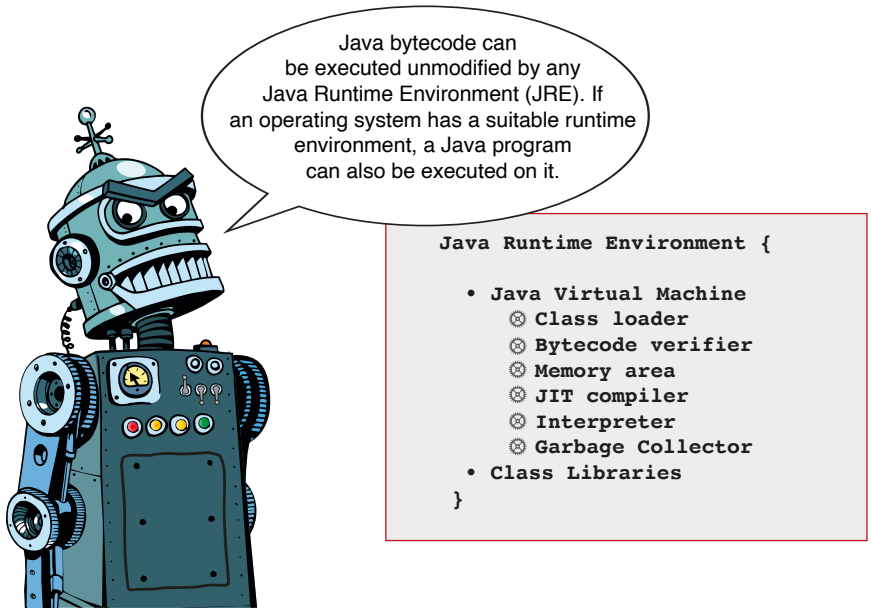


Figure 21.13: The bytecode can be executed unmodified by any virtual machine.

The availability of the virtual machine for different computer hardware and operating systems helps Java programs achieve their high portability. To run a Java program on any computer, it requires only a special startup script that the developer must supply.

21.10 Literature

Wikipedia: Java bytecode instruction listings; *https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings*

Wikipedia: HotSpot; *https://de.wikipedia.org/wiki/HotSpot*
Bernhard Steppan: "Packaging artist"; code wrapper for Java"=applications; *https://shop.heise.de/katalog/verpackungskunstler-2*

21.11 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/runtime_environment
There you will also find the solutions to the exercises.

22 Class Libraries

22.1 Introduction

Java class libraries contain ready-made classes for a wide range of applications. They extend the performance range of the Java language and increase the productivity of a software developer. This chapter explains how class libraries are structured and introduces you to the most important classes of the Java Standard Edition with examples.

Figure 22.1: Class libraries have many similarities with real libraries.

22.2 Overview

Class libraries are collections of proven programming solutions. They consist of thematically related classes, for example, for programming graphical interfaces or accessing databases. However, class libraries do not contain independent programs, but only building blocks for your programs. The similarity to classical libraries is obvious: The author of a class makes its solution available like a book to other programmers. The programmers »borrow« , so to speak, the class for their purposes. Not only the borrowing reminds of classical libraries, also the thematic framework. Therefore it came to the designation »class library«.

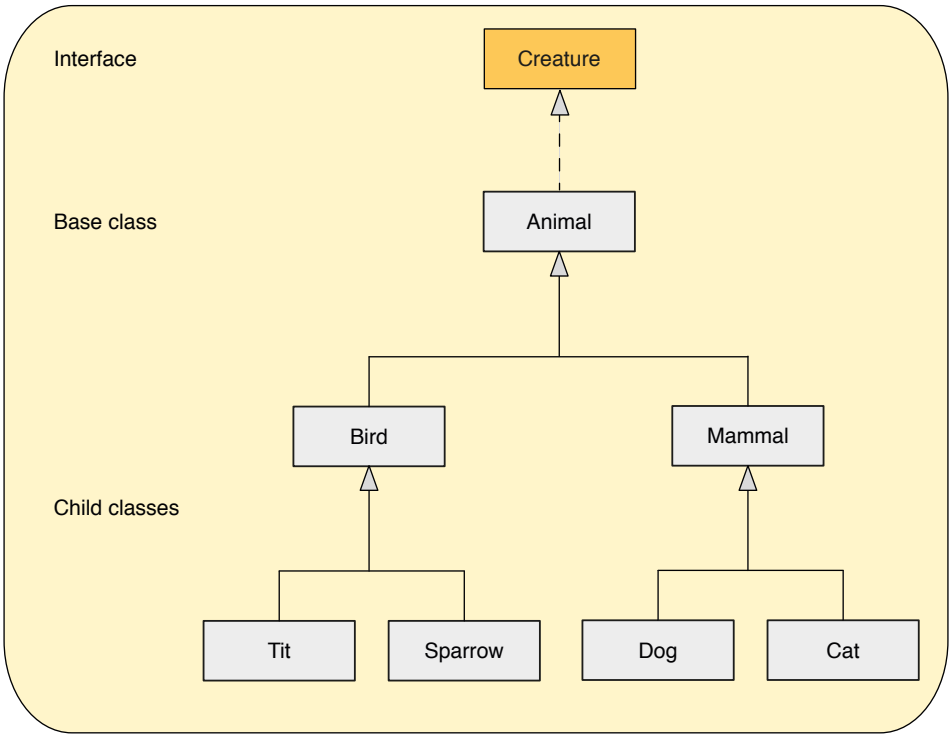


Figure 22.2: Class libraries are organized tree-like and thematically.

The first Java class libraries were very much organized using inheritance. Modern class libraries, on the other hand, consist of loosely coupled classes that are now only thematically related. By the mixture of abstract classes, interfaces and concrete as well as generic classes Java class libraries can be arranged fastidiously. One or several interfaces as well as basis classes form the basis for classes derived from it. As diagram a class library reminds of a tree standing on the head. Programmers therefore refer to such graphics as a class tree (Figure 22.2).

22.2.1 Areas of Application

Class libraries exist for every conceivable use case. For example, libraries exist for mathematical computation, image processing, and pattern recognition.

22.2.2 Reuse

You use these class libraries as a construction kit and pick out the building blocks that make sense for your program. You can focus on essentials in your program thanks to these classes, increasing your productivity. You simply download class libraries from the provider. Then you configure your development environment Eclipse so that you can access the desired classes with your programs.

22.2.3 Documentation

Documentation exists for all libraries in the form of JavaDoc. JavaDoc is directly available in an integrated development environment like Eclipse via the online help. Since JavaDoc is often a bit too sparse, other literature exists for many libraries. For example, Oracle offers further technical documentation via the Internet: *https://www.oracle.com/technetwork/java/javase/documentation/index.html*

22.2.4 Language Extension

Java, like any object-oriented language, is structured so that class libraries supplement a relatively small language core (Part II, »Java Language«). Any Java developer can also compile their own classes into libraries for their own use or for others. Due to the fact that Java has been around for over 20 years, thousands of Java classes have been created over the years.

22.2.5 Types of Class Libraries

Many software houses and consulting firms offer paid-for Java class libraries. Far more popular are open source libraries provided by many organizations such as Apache and Eclipse. Java class libraries provided by Oracle are now open source as well.

22.3 Java Standard Edition

Java is divided into three editions: Java Standard Edition (JSE), Java Enterprise Edition (JEE), and Java Micro Edition (JME). The focus of this book is on the Java Standard Edition. The libraries of this edition form the backbone of any Java program. They are part of

the Java runtime environment. The Java Standard Edition class libraries have been organized into modules since Java 9. Many modules that you can find on the Java homepage are marked as »deprecated«. This means that you should not use them anymore, because the Java manufacturer will remove them from the Java core. The modules that will last are the ones you see in Figure 22.3, which will remain relatively stable over the next few years.

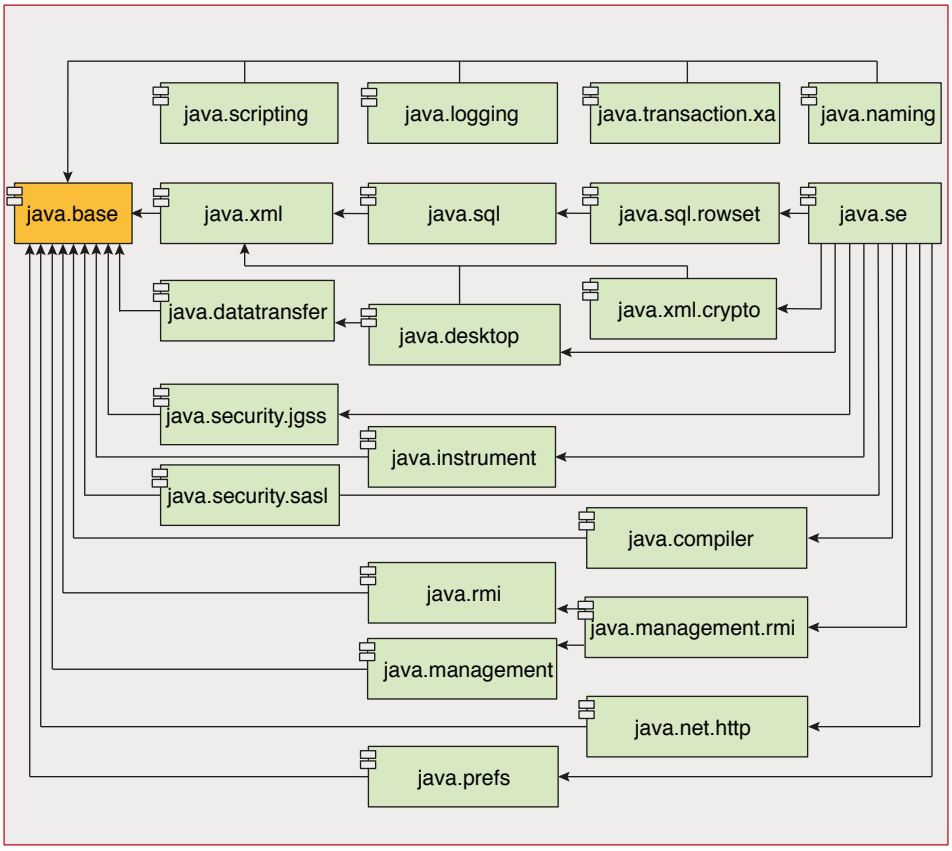


Figure 22.3: The modules of the Java standard library.

As you can see from the graphic, the »threads« converge at the *java.base* module. As the name implies, this module directly or indirectly forms the basis of all other modules. All other modules depend directly or indirectly on *java.base*.

22.3.1 Base Classes

The main package of the *java.base* module is *java.lang*. The *java.lang* package of the *java.base* module is the Java language library. Its classes form the basis for all other Java classes. They take a special position among all Java libraries, because they are loaded automatically. In addition, they are so strongly bound to the language core that certain language

rules are overridden for them. You can see what this means by looking at the different classes.

Class »Object«

The most important class of the *java.lang* package is the class with the somewhat strange name *Object*. All Java classes you create have this class as a superclass in the inheritance hierarchy (Figure 22.4).

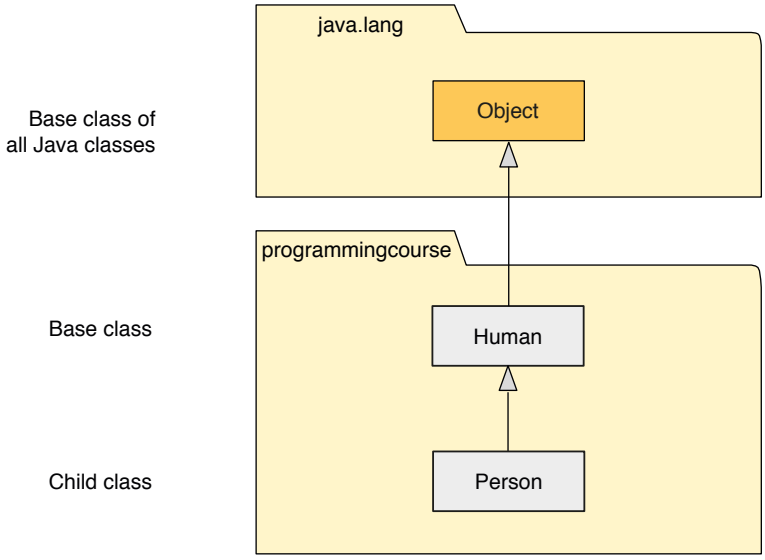


Figure 22.4: The class »Object« is the basis of all Java classes.

Now you are probably thinking that you have never seen this class before, neither in a class diagram nor in source code. You can't, because the class is always automatically present. It is inherited by force, so to speak. What is the point of all this? The reason is that some methods useful for all Java classes are to be inherited via the base class *Object*.

For example, the *Object* class has the *clone()* method and the *toString()* method. You use the *clone()* method to clone the existing object, and the *toString()* method to output a value for the object in the form of a string. To illustrate this using the *toString()* method, here's an example:

Let there be a hierarchy of three classes. The class *Being* shall again serve as the basis of the class *Human*. In Figure 22.4 you will find *Object* as the base class. An object of this class is automatically created in the background, although it does not appear in the source code of the base class (Listing 22.1). The class consists of only one attribute *name*, and a corresponding constructor, as well as one method to get the name and one to set it.

Listing 22.1: The base class »Human« extends the base class »Object« without any statement.

```
1 package programmingcourse;
2
```

```
3  public abstract class Human {
4
5    // The name of the person:
6    String name;
7
8    //Constructor with parameter name:
9    public Human(String name) {
10     this.name = name;
11   }
12
13   // Getter:
14   public String getName() {
15     return name;
16   }
17
18   // Setter:
19   public void setName(String name) {
20     this.name = name;
21   }
22 }
```

The *person* class derived from the *human* class consists of only a constructor and a method *toString()* (listing 22.2). The implemented method *toString()* overrides the base method of the superclass *Object*.

Listing 22.2: The derived class »Person«.

```
1  package programmingcourse;
2
3  class Person extends Human {
4
5    // Constructor:
6    public Person(String name) {
7      super(name);
8    }
9
10   // Override toString()
11   public String toString() {
12     return " ist eine Person";
13   }
```

You can run the program using the *ObjectDemo* class (Listing 22.3).

Listing 22.3: The class »ObjectDemo«.

```
1  package programmingcourse;
2
3  public class ObjectDemo {
4
5    public static void main(String[] arguments) {
6      Person julia = new Person("Julia");
7      System.out.print(julia.getName() + julia.toString());
```

```
8     }
9  }
```

The program generates the output:

```
Julia is a person
```

Because the base class *Person* overrides the *toString()* method with its own implementation, the program produces meaningful output. If you delete the *toString()* method in the class *Person*, you get the object ID as output. It shows that the base class *Object* has been forcibly inherited and methods like *toString()* are consequently part of every Java class. If you want to change the behavior of this method, you must override it.

Class »String«

To use strings in programs, the corresponding »char« data types are usually not flexible enough (Chapter 8, »Primitive Data Types«). As an alternative, a class called *String* is available in the package *java.lang*. Objects of type *String* can be declared and created quite easily (Figure 22.5).

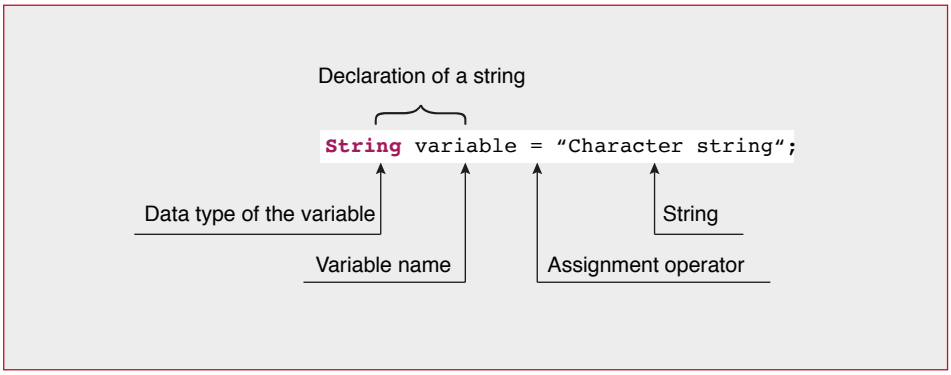


Figure 22.5: Declaration, creation and assignment of a string.

Listing 22.4 shows the difference from the normal creation of an object. As the example in lines 9 and 10 shows, new objects of type *String* can be created conventionally with the new operator, just like other objects. To do this, you declare the variable in the first step, create it with the new operator in the second step, and finally assign a string to it. But this would be very cumbersome. It is much faster to combine declaration, creation and assignment in one statement, as shown in line 13.

Listing 22.4: A first example of using strings.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  public class StringDemo {
6
```

```
7    public static void main(String[] arguments) {
8
9      String professor = new String(); // Creation
10     professor = "Professor Roth"; // Assignment
11
12     // Declaration + creation + assignment:
13     String robot = "Robert";
14
15     System.out.println("Professor: " + professor);
16     System.out.println("Robot: " + robot);
17   }
18 }
```

Since processing characters and larger text is enormously important to any program, strings are also extremely significant to Java programming. The class *String* is therefore very extensive and powerful. Table 22.1 gives an impression of the performance diversity. It gives an overview of the constructors of the class *String*.

Table 22.1: Constructors of the class »String«.

Constructor	Meaning
String()	Creates an empty string.
String(byte[] bytes)	Creates a string and converts the byte array.
String(byte[] ascii, int hibyte)	This method is deprecated and should not be used.
String(byte[] bytes, int offset, int length)	Generates a string from the byte array. *offset* specifies the starting point within the array and *length* the length.
String(byte[] ascii, int hibyte, int offset, int count)	This method is deprecated and should not be used.
String(byte[] bytes, int offset, int length, String enc)	Generates a string from the byte array. *offset* specifies the starting point within the array, *length* the length and *enc* the encoding.
String(byte[] bytes, String enc)	Generates a string from the byte array. *enc* specifies the encoding.
String(char[] value)	Generates a string from the char array.
String(char[] value, int offset, int count)	Generates a string from the char array. *offset* specifies the starting point within the array and *count* the number of characters.
String(String original)	Creates a copy of the passed string.
String(StringBuffer buffer)	Creates a string from a StringBuffer.

As you can see from the table, *only* are constructors of the class *String*, methods that you can use to create a string. Some of these constructors are *deprecated* and should no longer be used. They date from the early days of Java and may soon be removed from the library.

The remaining constructors are available if you want to create a new string. In addition, there are a great many class methods available to you, which I won't list here for space reasons – but that's not necessary, since the Java documentation for all class libraries is either included in your development environment or available on the Internet.

All graphical user interfaces handle strings practically exclusively. Even numeric input is initially received by a Java program as a string. In order to be able to calculate with them within the program, they must of course be converted back into integers or decimal numbers. Special wrapper classes serve as a bridge for the conversion: the simple class types.

Wrapper Classes

Chapter 8, »Primitive Data Types«, has introduced you to eight primitive data types. Variables based on these data types are not objects and therefore do not have methods. However, sometimes it would be necessary to convert or modify these values. To be able to use these simple data types like objects, there are eight matching classes that wrap them – they wrap a simple data type with an object-oriented shell.

Table 22.2: Simple class types wrap primitive data types.

Constructor	Meaning
Byte	byte
Short	short
Integer	int
Long	long
Double	double
Float	float
Boolean	boolean
character	char

One use case for wrapper classes is to convert strings to numeric values and vice versa. As mentioned earlier, Java graphical interfaces process all numeric values in string format. To be able to calculate with the numerical values, it is necessary to convert the strings that the user has entered back into a numerical format (Figure 22.6).

After that, mathematical calculations can be performed with the numerical values, which for obvious reasons are not possible with strings (sin(), cos(), addition, etc.). After the calculation the situation occurs again that the user wants to see the final result. In order to display the numerical values, they have to be converted to a string again (Figure 22.7).

```
public int stringToInt(String text) {
  return Integer.valueOf(text);
}
```

Figure 22.6: Conversion of a string to a number.

The conversion cannot be done as it was shown in chapter 23, »Rules«, in section 23.6, »Type Conversion«, using primitive data types. These primitive data types, while different

in value range, are very similar to each other: they are simple memory cells, do not have methods, and thus are not classes. Therefore, the type conversion of a simple data type into a complex data type such as a class can only be done using wrapper classes.

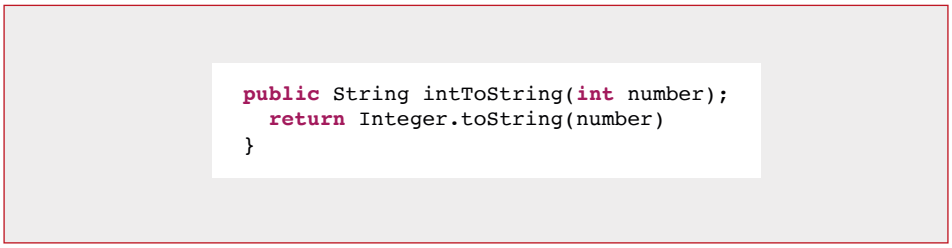


Figure 22.7: Conversion of a number to a string.

A sample program shows what happens when you try to calculate using a string (Listing 22.5). The program initially assigns a value of 200 to a number created as *String* and then increments it.

Listing 22.5: A first example of string conversion.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  class StringConversionDemo {
6
7    public static void main(String[] arguments) {
8
9      String text; // Declaration
10     int number; // Declaration
11
12     // 1st trial without conversion:
13
14     // Assign the value 200:
15     text = "200";
16
17     //  Pseudo calculation: text++ nicht erlaubt:
18     text = text + 1;
19
20     // The result is wrong:
21     System.out.println("Der Wert lautet: " + text);
22
23     // 2st trial with conversion:
24
25     // Assign the value 200:
26     text = "200";
27
28     // Convert String -> int:
29     number = Integer.valueOf(text);
30
```

```
31      // Increment number = number++:
32      number = number + 1;
33
34      // The result is correct:
35      System.out.println("The value is: " + number);
36    }
37  }
```

The program produces the following output:

```
The value is: 2001
```

In the variant, the program converts the number stored as *string* to a text, then increments it and outputs it afterwards. It appears this time:

```
The value is: 201
```

Without the help of the wrapper class *Integer* it would not have been possible to make this conversion and calculate the correct value.

Class »StringBu§er«

Concatenating strings with a plus sign is a wonderfully simple way to concatenate strings - but it has a reputation for being very slow. In addition, the *String* class has a number of limitations. As a consequence, wherever strings need to be joined and modified frequently, you should use the *StringBuffer* class (Listing 22.6).

Listing 22.6: The program »StringBufferDemo«.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  class StringBufferDemo {
6
7    public StringBuffer string() {
8      StringBuffer str = new StringBuffer("String");
9      for (int i = 0; i <= 10; i++)
10       str.append(str);
11     return str;
12   }
13
14   public static void main(String[] arguments) {
15     StringBufferDemo stretch = new StringBufferDemo();
16     StringBuffer str = new StringBuffer();
17     str = stretch.string();
18     System.out.println("Result: " + str);
19   }
20 }
```

The program *StringBufferDemo* shows an application of the class *StringBuffer*. It concatenates a string in a loop and then outputs it to the console. This process runs very fast. The result is a character field.

22.3.2 Class »System«

Along with the String class and the wrapper classes, the *System* class is a linchpin of any Java program. It has a variety of tasks, including:

- Character input and output to the console
- Calling the garbage collector
- Program termination
- Setting basic system settings
- Query basic system settings

You have already seen the output of characters in many examples in the book. The use of the garbage collector was explained in chapter 21, »Runtime Environment«, section 21.4.5, »Garbage Collector«. What is new, however, is that the program is terminated by a method of the class *System*. The call *System.exit(status);* is sufficient for this.

You pass the state of the program via the int variable *status*. You should pass null if the program terminated normally. You can also output a defined other value to indicate that the program terminated after a critical state.

Another often very useful function of the *System* class is to read in or out system settings. How reading out system information (system properties) works in principle is shown in Listing 22.7.

Listing 22.7: Read some system properties.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  class SystemInfoDemo {
6
7    public static void main(String[] arguments) {
8
9      System.out.println(
10         "Java version: " + System.getProperty("java.version") + "\n" +
11         "Java home: " + System.getProperty("java.home") + "\n" +
12         "JVM version: " + System.getProperty("java.vm.version") + "\n" +
13         "JVM name: " + System.getProperty("java.vm.name") + "\n" +
14         "Operating system: " + System.getProperty("os.name") + " " +
15           System.getProperty("os.version") + "\n" +
16         "User name: "+ System.getProperty("user.name") + "\n" +
17         "User home: "+ System.getProperty("user.home"));
18   }
19 }
```

To do this, you use the access method *getProperty()*, which you pass a specific string as a parameter. These strings are predefined in the Java documentation. For example, to find out the name of the operating system under which the Java program is running, use the property *os.name*. The program produces different output depending on the installed Java and operating system version. On my development machine, the output looks like this:

```
Java version: 13.0.1
Java-Home: /Library/Java/JavaVirtualMachines/jdk-13.0.1.jdk/Contents/Home
JVM version: 13.0.2+8
JVM name: Java HotSpot(TM) 64-bit Server VM
Operating system: Mac OS X 10.15.3
User name: bsteppan
User directory: /Users/bsteppan
```

Using these basic system settings gives you important information that your program can use. For example, it is able to store user-specific data in the user's directory based on the user name.

Another application of the *System* class is time measurements. To perform these, use the method *currentTimeMillis*. It returns a long type that you can use elegantly as a stopwatch. In listing 22.8 you can see the measurement of the speed of your computer. It is not a complete measurement, but mainly measuring the speed of the processor and the screen output.

Listing 22.8: An example of a time measurement.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  public class TimeDemo {
6
7    public static void main(String[] args) {
8
9      long startTime = System.currentTimeMillis();
10
11     for (int run = 0; run <= 1000; run++) {
12       System.out.println("Run no.: " + run);
13     }
14
15     long endTime = System.currentTimeMillis();
16     long totalTime = endTime - startTime;
17     System.out.println("Total time of all runs: " + totalTime + " ms");
18   }
19 }
```

The program *TimeDemo* starts by calling the *currentTimeMillis* method of the class *System* and storing the return value in a variable named *startTime*. After that, the program performs a calculation in a longer loop. The program measures the time used for this after the loop and assigns it to the variable *stopTime*. Then the program forms the difference of both times and outputs the calculated value. My current fastest computer outputs the loop as follows:

```
Pass number: 1
(...)
Pass number: 1000
Time of all passes: 12 ms
```

22.3.2.1 Class »Runtime«

The class *Runtime* has, among other things, the task of starting other programs from Java programs. This works as shown in Listing 22.9.

Listing 22.9: The extended example of a program startup.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  import java.io.BufferedReader;
6  import java.io.IOException;
7  import java.io.InputStreamReader;
8
9  class RuntimeDemo {
10
11   @SuppressWarnings("deprecation")
12   public static void main(String[] arguments) {
13
14     Process process = null;
15     try {
16       //Unix
17       process = Runtime.getRuntime().exec("ls -l");
18       //Win9x
19       //process = Runtime.getRuntime().exec("command /c dir");
20       //WinNT
21       //process = Runtime.getRuntime().exec("cmd /c dir");
22       BufferedReader input = new BufferedReader(
23           new InputStreamReader(process.getInputStream()));
24       String output;
25       while ((output = input.readLine()) != null) {
26         System.out.println(output);
27       }
28     }
29     catch (IOException fehler) {
30       System.out.println(fehler);
31     }
32   }
33 }
```

The program is able to list the directory contents. For this purpose it uses a console. For a Unix terminal it is sufficient to call the corresponding Unix command. With the Windows family the treatment is somewhat different. Here the DOS prompt must be started either with *cmd* or with *command*.

Unlike the first variant of the program, this time this example evaluates the return object of type *Process* to take the output of the console. The program then prints the characters from this output stream back to the console. For example, on my Mac (Unix), the program produces the following information:

```
total 0
drwxr-xr-x 3 bsteppan staff 96 Feb 16 15:55 bin
```

```
drwxr-xr-x 4 bsteppan staff 128 Feb 17 19:19 src
```

22.3.3 Threads

In some situations it is necessary for a program to do not only one task, but several in parallel. In this program then several threads of action run, so-called *threads*. To avoid the situation in which every Java programmer has to develop such difficult concurrent programs from scratch, there is support for this in the Java class library as well.

The help comes from a class called *Thread* and an interface called *Runnable*. The class *Thread* implements this interface and provides every Java programmer with important functions to start, stop and synchronize mini-programs within their program. Here is an example, which consists of three classes. The class *ThreadDemo* is the main program with the start method *main* (Listing 22.10).

Listing 22.10: The main program starts the »sewing machine«.

```
1  //Beispielprogramme/Klassenbibliotheken
2
3  package programmingcourse;
4
5  public class ThreadDemo {
6
7    public static void main(String[] arguments) {
8      Thread up = new Thread(new RightThread());
9      up.start();
10     Thread down = new Thread(new LeftThread());
11     down.start();
12   }
13 }
```

The classes *LeftThread* and *RightThread* both have the class *Thread* as their base class. The *LeftThread* class outputs a lowercase character. The character points to the left and gives the class its name (Listing 22.11).

Listing 22.11: This is the one thread ...

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  public class LeftThread extends Thread {
6
7    public void run() {
8      for (int i = 0; i < 100; i++) {
9        System.out.print("<");
10     }
11   }
12 }
```

The class *RightThread* outputs a greater-than character. The character points to the right, hence the name of the class (listing 22.12).

Listing 22.12: ... and that is the other.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  public class RightThread extends Thread {
6
7    public void run() {
8      for (int i = 0; i < 100; i++) {
9        System.out.print(">");
10     }
11   }
12 }
```

If the program were to execute the method *run* of the two classes strictly alternately, an interesting knitting pattern of less-than and greater-than characters should result. In this case, however, the processing is concurrent and achieves an effect familiar to modern operating systems:

Several programs can be executed side by side, even though only one CPU is available. In operating systems this is called multitasking. Multitasking within a program is called multithreading. Thread programming is based on a conscious illusion: It is an apparent parallel processing. The program switches from one thread to another at more or less regular intervals, as the output of the program shows.

```
>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<<<
```

Listing 22.13: This program outputs the contents of a file.

```
1  //Examples/Class_Libaries
2
3  package programmingcourse;
4
5  import java.io.BufferedReader;
6  import java.io.FileNotFoundException;
7  import java.io.FileReader;
8  import java.io.IOException;
9
10 public class FileReaderDemo {
11
12   public static void main(String[] arguments) {
13
14     BufferedReader recipe = null;
15     String text;
16
17     try {
18       recipe = new BufferedReader(new FileReader("Filet.txt"));
19     } catch (FileNotFoundException exception) {
20       System.out.println("Error: " + exception);
21     }
22     try {
23       while ((text = recipe.readLine()) != null) {
24         System.out.println(text);
25       }
26     } catch (IOException exception) {
27       System.out.println("Error: " + exception);
28     }
29   }
30 }
```

The result of this action is a stream of characters, which is of little use because the point of the program is to *output* text. For this reason, the stream of characters must be passed to an object of type *BufferedReader*. This object is able to *output* the characters again via a while loop, which is done as follows:

```
Filet With Salad
Take a 3 cm thick filet, salt and pepper it from one side,
fry it one side in a pan for 3 minutes,
turn it over, salt and pepper it again,
fry it again for 3 minutes and serve it with salad.
```

Class »FileWriter«

Just as smoothly and quickly as a few lines of Java code read a file, creating a new file works. To do this, create an object of the type *FileWriter* and pass the name of the file to its constructor (Listing 22.13).

Listing 22.14: This program writes the contents of a text to a file.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  import java.io.FileWriter;
6  import java.io.IOException;
7
8  public class FileWriterDemo {
9
10   public static void main(String[] arguments) {
11
12     FileWriter recipe = null;
13
14     // Try to write the file:
15     try {
16       // File names:
17       recipe = new FileWriter("Filet.txt");
18
19       // Der Text in Form eines Strings:
20       String text = "Filet␣With␣Salad\n"
21           + "Take␣a␣3␣cm␣thick␣filet,␣salt␣and␣pepper␣it␣from␣one␣side,\n
                 "
22           + "fry␣it␣one␣side␣in␣a␣pan␣for␣3␣minutes,\n"
23           + "turn␣it␣over,␣salt␣and␣pepper␣it␣again,\n"
24           + "fry␣it␣again␣for␣3␣minutes␣and␣serve␣it␣with␣salad.";
25       recipe.write(text);// String -> file
26       recipe.close();// Close file
27     } catch (IOException exception) {// Error
28       System.out.println("Error:␣" + exception);
29     }
30   }
31 }
```

The object, called *file* in this program, has several methods. With *write* you write text to this file, and with *close* you close the file again, so that it appears on the file system - there is nothing more to program, except for the mandatory exception handling in case of write errors.

22.3.5 Properties

Sometimes basic settings of a program need to be read and stored permanently on disk. On Windows, programs use ini files or store their information in the registry. To be independent of a particular operating system, the Java inventors came up with a different concept: the concept of property files.

These property files have a trivial structure and consist of a key and one or more matching values (Figure 22.8). You can separate key and value(s) either by a space, by a colon, or by an equal sign. You read the values by calling a method of the *Properties* class and getting a string containing the value or list of values as return value.

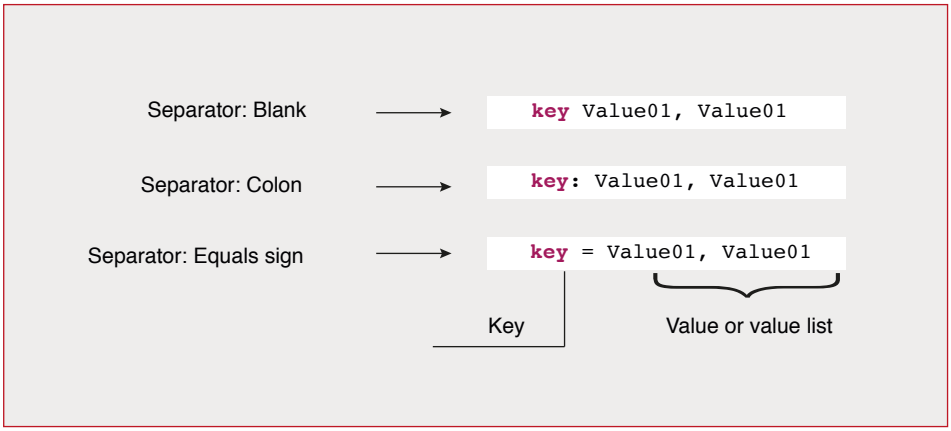


Figure 22.8: The structure of a properties file.

This sounds a lot like the stream examples, so it's not surprising that the property class is built on top of the stream library. Unlike the character streams already shown, properties allow structured files to be read in and out. How this works is shown in the following example (Listing 22.15).

Listing 22.15: This program reads the contents of a structured file.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  import java.io.FileInputStream;
6  import java.io.IOException;
7  import java.util.Properties;
8
9  public class PropertiesDemo {
10
11   public static void main(String[] arguments) {
12
13     Properties weekdays = new Properties();
14     String propertyName = null;
15
16     // Try to open the file
17     try {
18       if (arguments[0].equals("en"))
19         propertyName = "Week_en.properties";
20       else
21         propertyName = "Week_de.properties";
22       weekdays.load(new FileInputStream(propertyName));
23     } catch (IOException exception) {
24       System.out.println(exception);// Error
25     }
26
```

```
27      // File output:
28      System.out.println(weekdays.getProperty("Monday", "Montag"));
29    }
30  }
```

The program creates a new object named *weekdays* of type *Properties* and then loads the contents of the file *Week_en.properties* or *Week_de.properties*. Which of the two files the program reads depends on a command line parameter. Using the getter function *getProperty* you can specifically read the contents of the value list (line 22).

Listing 22.16: The contents of this properties file are the German weekdays.

```
1  Monday = Montag
2  Tuesday = Dienstag
3  Wednesday = Mittwoch
4  Thursday = Donnerstag
5  Friday = Freitag
6  Saturday = Samstag
7  Sunday = Sonntag
```

22.3.6 Container Classes

Another important area of auxiliary classes is the inclusion of arbitrary values and objects. While simple arrays (Chapter 11, »Arrays«) were »only« semi-dynamic, the powerful container classes *Vector* and *Hashtable* allow to create any number of objects dynamically.

The example clearly shows the difference between arrays and vectors. At the beginning of the *main* method, the program creates a string array with the six people of the programming course. These strings are passed to a vector, which does not »know« how many elements to include in its member list.

Listing 22.17: A vector manages objects completely dynamically.

```
1  //Examples/Class_Libraries
2
3  package programmingcourse;
4
5  import java.util.Vector;
6
7  public class VectorDemo {
8
9    public static void main(String[] arguments) {
10
11     int index;
12     String programmingCourse[] =
13       new String[] {"Karsten", "Anna", "Julia",
14                     "Lukas", "Florian", "Robert"};
15
16     Vector<String> membersList = new Vector<String>();
```

```
17
18      System.out.println("The first names of the members "
19          + "of the programming course are:");
20      for (index = 0; index < programmingCourse.length - 1; index++) {
21        membersList.addElement(programmingCourse[index]);
22        System.out.print(membersList.get(index) + ", ");
23      }
24      membersList.addElement(programmingCourse[index]);
25      System.out.print(membersList.get(index));
26    }
27  }
```

A simple for loop drops new members into the member list using the *addElement* method. Finally, the program uses the *get* method to output the strings from the member list again. The *get* method wants to get the index of the element it should output as a parameter. The program produces the following output:

```
The first names of the members of the programming course:
Karsten, Anna, Julia, Lukas, Florian, Robert
```

22.3.7 Abstract Windowing Toolkit

The Abstract Windowing Toolkit (AWT) and Swing are part of Java. For a long time they were the standard libraries for programming graphical user interfaces (GUI) of Java programs. Only later appeared other GUI libraries like SWT and JFace as well as JavaFX. In the early days of Java, the Abstract Windowing Toolkit was mainly used to completely design graphical user interfaces (Figure 22.9). This by no means includes only the interface components, but also character sets, colors, and message processing.

GUIs developed with AWT look exactly like native GUIs because AWT uses the operating system's functions for rendering. Unfortunately, the AWT contained many bugs and shortcomings. This led to the fact that today nobody develops graphical user interfaces that consist only of AWT components.

Today one designs Java GUIs either from a combination of AWT and the library Swing or with JavaFX. As a further alternative, the Eclipse consortium offers the Standard Widget Toolkit library with JFace and the Eclipse Rich Client Platform. The Eclipse development environment is based on this GUI platform.

Despite the fact that the GUI components of the AWT are practically irrelevant today, the AWT continues to be of great importance in the Java world. It is needed to use character sets or colors. Other important tasks are layout managers and event handling within Java programs.

Event Handling

The programming of graphical user interfaces is event-driven. This means that every mouse click, keystroke, moving, zooming in and out of windows triggers events. There are three participants in event control:

- Trigger

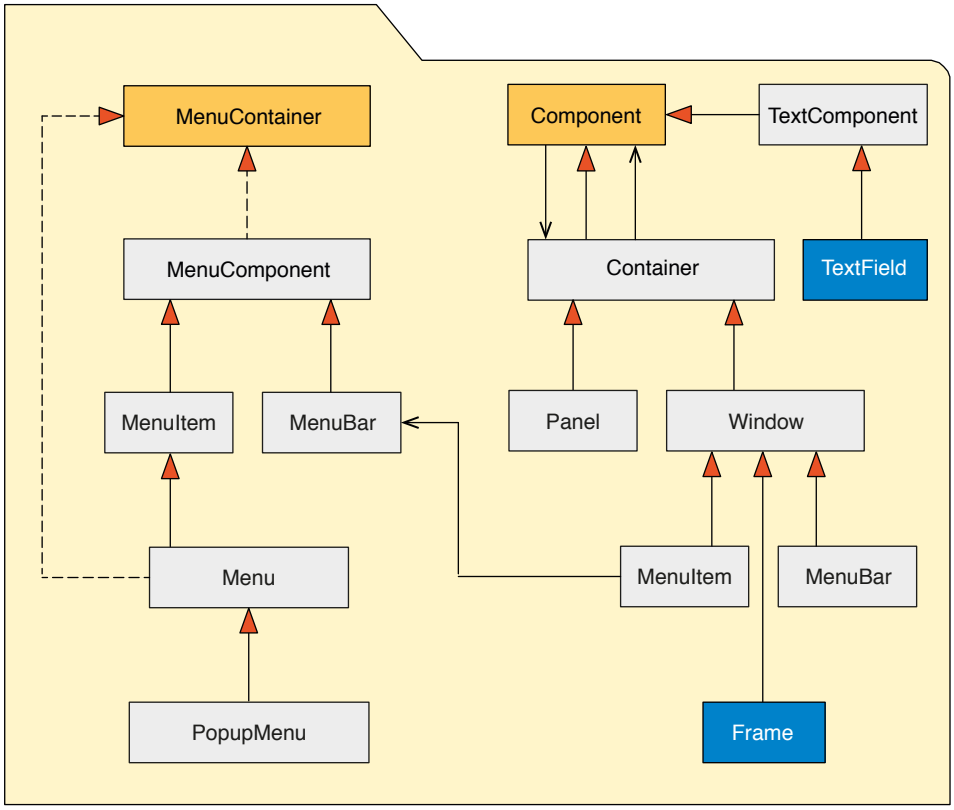


Figure 22.9: The main GUI classes of the AWT library.

- Message
- Receiver

Figure 22.10 shows the process in four steps. In step 1, the user enlarges the program's window (*trigger*). As a result, the Java Runtime Environment (JRE) emits a series of signals, for example, the message *ComponentResized*.

If the program (*Receiver*) wants to respond to this event specifically, it intercepts this message and then redraws the contents of the window. The following example (Listing 22.18) shows how this works again on an AWT window. The constructor of the program window first creates an anonymous class based on the AWT class *ComponentAdapter*. This AWT class allows to monitor the events of components, which includes the main window. This works because the new window class *EventDemo* inherits from *Frame*, which in turn derives from the AWT class *Component*.

Listing 22.18: The main program creates a new AWT window.

```
1 //Examples/Class_Libraries
2
3 package programmingcourse;
```

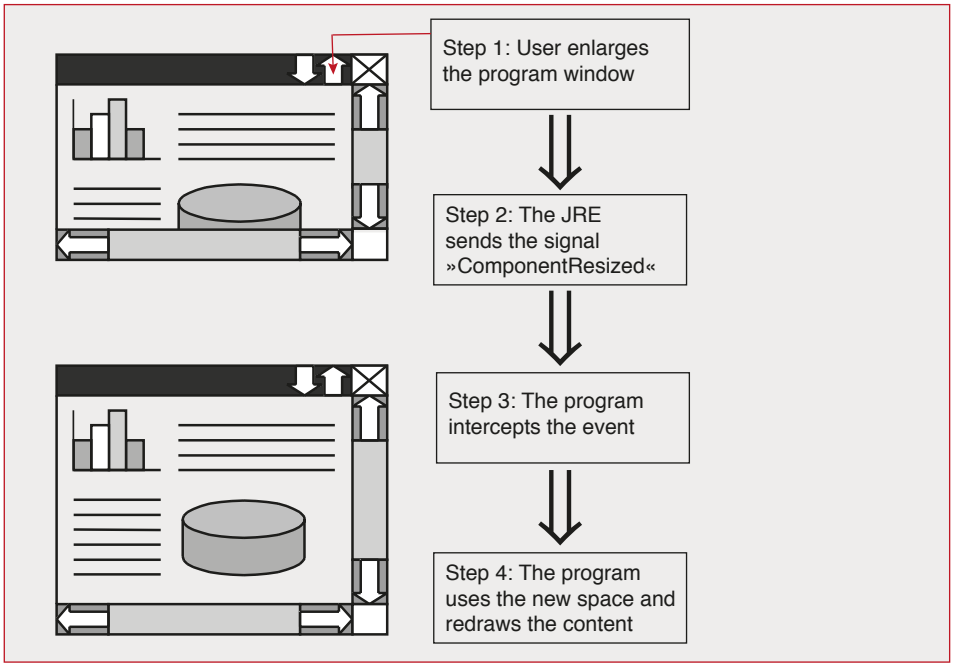


Figure 22.10: Event handling within a program.

```
4
5  import java.awt.BorderLayout;
6  import java.awt.Frame;
7  import java.awt.TextField;
8  import java.awt.event.ComponentEvent;
9  import java.awt.event.WindowAdapter;
10 import java.awt.event.WindowEvent;
11
12 public class EventDemo extends Frame {
13
14   private static final long serialVersionUID = 1L;
15
16   public EventDemo() {
17     addWindowListener(new WindowAdapter() {
18       public void windowClosing(WindowEvent e) {
19         dispose();
20         System.exit(0);
21       }
22     });
23     setSize(500, 200);
24     setLayout(new BorderLayout());
25     TextField information = new TextField("");
26     add("Center", information);
27
28     addComponentListener(new java.awt.event.ComponentAdapter() {
```

```
29
30        public void componentMoved(ComponentEvent e) {
31          information.setText("Window moved");
32        }
33
34        public void componentResized(ComponentEvent e) {
35          information.setText("Window resized");
36        }
37      });
38    }
39
40    public static void main(String[] arguments) {
41      EventDemo eventDemo = new EventDemo();
42      eventDemo.setVisible(true);
43    }
44  }
```

Since the program wants to react individually to the *componentMoved* and *componentResized* events, it is necessary to override the methods with the same names. The program replaces the implementation of the base class with its own. However, the implementation of the methods only contains a message showing that the program has intercepted the message. It does not process the message any further.

In larger event-driven programs, many of these actions run through the basic *paint* method, which must be overridden if you want to redraw the graphical contents of the window itself. However, if this is not the case and you are using interface components such as text boxes, lists, or buttons, you should also react to resizing a window. This is done using layout managers.

Use of Layout Managers

Layout managers are used to react to resizing a window according to a certain strategy. The strategy determines what arrangement the interface components such as text boxes, buttons, etc. get after the realignment. In primitive GUI libraries, one arranges individual GUI components by absolute coordinates. As a result, windows cannot be resized or do not respond as expected. Figure 22.11 shows an example of this. Here, as in all window systems, the coordinate origin is at the top left. When the user enlarges the window, the interface components remain in the upper left corner (coordinate origin) and do not change their size.

In Java programs, the coordinate origin is also at the top left, but people almost exclusively use relative coordinates and maximum and minimum sizes of components. The figure shows what this does when additional layout managers take over the task of automatically aligning all components. The size of the text box has been adjusted, and we have aligned its position according to the new window size (Figure 22.12).

Two layout managers are very important for understanding many Java programs: »Border Layout« and »GridBag Layout«.

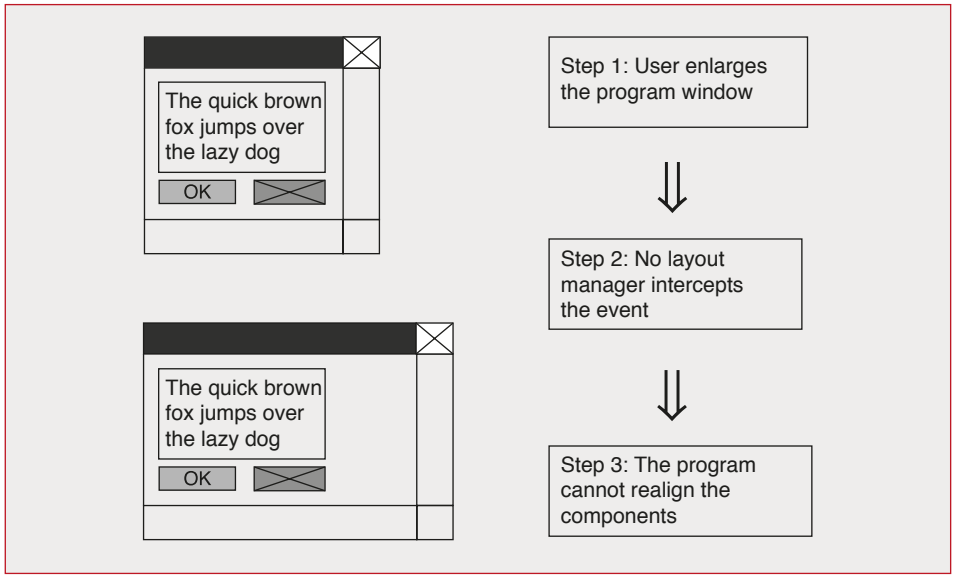


Figure 22.11: Without layout manager the components do not realign.

Border Layout

The Border Layout aligns the user interface components to the border of the window. There are five areas to which a component can be aligned:

- North: upper window border
- South: lower window border
- West: left window border
- East: right window border
- Middle: center of the window

How these parameters are used within a program is demonstrated by the example in Listing 22.19. The program is derived from the AWT base class *Frame* (Figure 22.9). The window contains five text fields. These five text fields are objects of type *TextField*. Each window of type *Frame* has a method *add(String name, Component comp)*. The method comes from the base class *Container*. It expects as parameters the desired alignment of the component and the object to be displayed.

Listing 22.19: Example of a border layout.

```
1 //Examples/Class_Libraries
2
3 package programmingcourse;
4
5 // Import AWT classes:
6 import java.awt.BorderLayout;
7 import java.awt.Frame;
8 import java.awt.TextField;
```

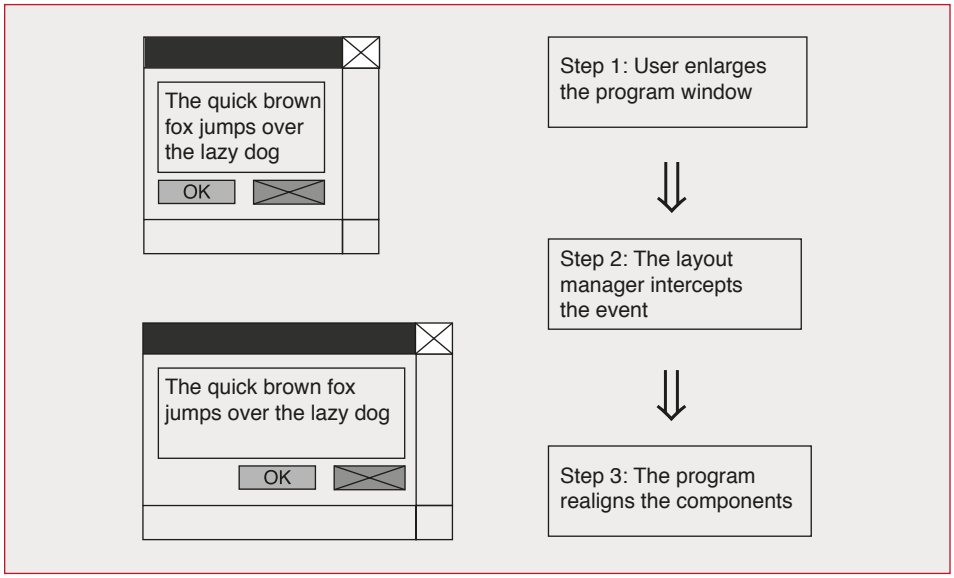


Figure 22.12: With a layout manager the components realign themselves.

```
 9  import java.awt.event.WindowAdapter;
10  import java.awt.event.WindowEvent;
11
12  public class BorderLayoutDemo extends Frame {
13
14    private static final long serialVersionUID = 1L;
15
16    public BorderLayoutDemo() {
17
18      addWindowListener(new WindowAdapter() {
19        public void windowClosing(WindowEvent e) {
20          dispose();
21          System.exit(0);
22        }
23      });
24
25      setSize(500, 300);
26      setLayout(new BorderLayout());
27
28      try {
29        add("North", new TextField("North"));
30        add("South", new TextField("South"));
31        add("West", new TextField("West"));
32        add("East", new TextField("East"));
33        add("Center", new TextField("Center"));
34      } catch (IllegalArgumentException exception) {
35        System.out.println("Wrong parameter: " + exception);
36      }
```

```
37    }
38
39    public static void main(String[] arguments) {
40      BorderLayoutDemo borderLayoutDemo = new BorderLayoutDemo();
41      borderLayoutDemo.setVisible(true);
42    }
43  }
```

Between lines 29 and 33 of the listing, you can see how the program adds the five text boxes with their alignment. Important: If one of the arguments causes problems, for example because an object is not initialized correctly or an alignment is specified that does not exist, the program sends an *IllegalArgumentException*. This does not necessarily have to be caught. However, if such an error occurs, it will cause the program interface to be invisible.

In Figure 22.13, you can see how the sample program is displayed. The layout manager correctly aligns all components to the border according to their parameter (*North, South, West, East, Middle*).

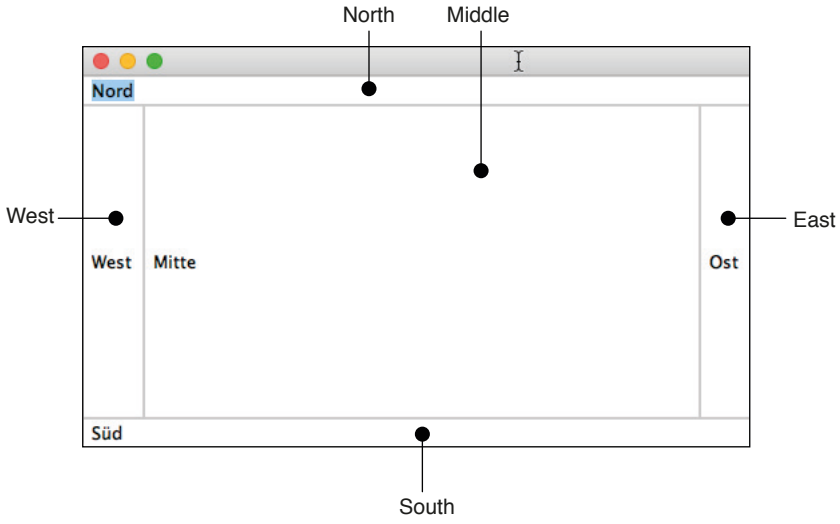


Figure 22.13: The border layout aligns components to the border.

GridBag Layout

The GridBag layout is one of the most flexible layouts. Because it is so flexible, it is diffi-cult to program and feared by Java developers because of its (seemingly) unpredictable behavior. Unlike the Border layout presented earlier, components positioned over a GridBag layout do not align with the border, but with an imaginary grid (hence the layout's name). Listing 22.20 shows an example of a program with five text fields that uses a GridBag layout.

Listing 22.20: Example of a GridBag layout.

```
1  //Examples/Class_Libraries
```

```
2
3  package programmingcourse;
4
5  // Import AWT classes:
6  import java.awt.Frame;
7  import java.awt.GridBagConstraints;
8  import java.awt.GridBagLayout;
9  import java.awt.Insets;
10 import java.awt.TextField;
11 import java.awt.event.WindowAdapter;
12 import java.awt.event.WindowEvent;
13
14 public class GridBagLayoutDemo extends Frame {
15   private static final long serialVersionUID = 1L;
16   GridBagLayout gridBagLayout = new GridBagLayout();
17   TextField textField00 = new TextField();
18   TextField textField10 = new TextField();
19   TextField textField20 = new TextField();
20   TextField textField01 = new TextField();
21   TextField textField02 = new TextField();
22
23   public GridBagLayoutDemo() {
24     this.addWindowListener(new WindowAdapter() {
25       public void windowClosing(WindowEvent e) {
26         dispose();
27         System.exit(0);
28       }
29     });
30     try {
31       initialize();
32     } catch (Exception e) {
33       e.printStackTrace();
34     }
35   }
36
37   private void initialize() throws Exception {
38     setSize(500, 200);
39     setLayout(gridBagLayout);
40     setBackground(java.awt.Color.lightGray);
41     textField00.setText("Cell (0|0), Height 1");
42     textField10.setText("Cell (1|0), Height 1");
43     textField20.setText("Cell (2|0), Height 2");
44     textField01.setText("Cell (0|1), Height 1");
45     textField02.setText("Cell (0|2), Height 1");
46     add(textField00, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
47       GridBagConstraints.CENTER, GridBagConstraints.NONE,
48         new Insets(0, 0, 0, 0), 0, 0));
49     add(textField10, new GridBagConstraints(1, 0, 1, 1, 0.0, 0.0,
50       GridBagConstraints.CENTER, GridBagConstraints.NONE,
51         new Insets(0, 0, 0, 0), 0, 0));
52     add(textField20, new GridBagConstraints(2, 0, 1, 2, 0.0, 0.0,
53       GridBagConstraints.CENTER, GridBagConstraints.VERTICAL,
```

```
54          new Insets(0, 0, 0, 0), 0, 0));
55      add(textField01, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0,
56        GridBagConstraints.CENTER, GridBagConstraints.NONE,
57          new Insets(0, 0, 0, 0), 0, 0));
58      add(textField02, new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0,
59        GridBagConstraints.CENTER, GridBagConstraints.NONE,
60          new Insets(0, 0, 0, 0), 0, 0));
61    }
62
63    public static void main(String[] arguments) {
64      GridBagLayoutDemo gridBayLayoutDemo = new GridBagLayoutDemo();
65      gridBayLayoutDemo.setVisible(true);
66    }
67  }
```

In Figure 22.14, you can see the program of the listing 22.20. The constructor of the window calls an initialization method in which the text boxes are created and positioned in a grid. From line 17 to 21, the program creates five text fields of type *TextField*. From line 46 to 60 the program assigns the text fields their position in the grid.

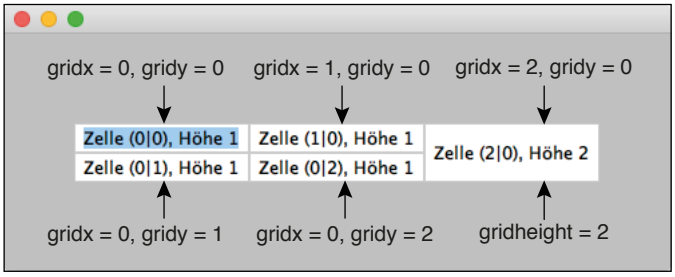


Figure 22.14: The GridBag layout aligns components in the form of a grid.

The grid consists of objects of type *GridBagConstraint*. The constructor of the class expects various parameters that are used to align the components (Table 22.3). Use *gridx* and *gridy* to specify at which grid position the component will be placed. For example, the coordinates (0|0) cause the component to be in the first row and in the first column. As with all Java graphics libraries, the coordinate origin is at the top left. So it is numbered from top left to bottom right.

By setting width (*gridwidth*) and height (*gridheight*) you specify the number of cells used by the component. This value is a fixed-point number that refers to the number of cells in a column or row. For example, in the case of text box 3 in the example program, the width is 1 but the height is 2. That is, the text box occupies two cells in height. The (*weightx*) and (*weighty*) parameters specify the horizontal and vertical weighting of the component. These parameters can be used to control how much of the free space within a window is reserved for the components when the size of the window changes.

Each component must be anchored at a specific location. The anchor point *anchor* determines in which direction the component expands when the window is resized. Similar to the Border layout, the following constants are available for anchoring: *NORTHWEST*, *NORTH*, *NORTHEAST*, *WEST*, *CENTER*, *EAST*, *SOUTHWEST*, *SOUTH* and *SOUTHEAST*.

Table 22.3 Parameter of the constructor of the class »GridBagConstraint«.

Data type	Identifier	Meaning
int	*gridx*	grid position x
int	*gridy*	grid position y
int	*gridwidth*	grid width
int	*gridheight*	grid height
double	*weightx*	x-weight
double	*weighty*	y-weight
int	*anchor*	anchoring (NW, W, SW, N, Z, S, NE, E, SE)
int	*fill*	fill type (without, horizontal, vertical, both)
insets	*insets*	space
int	*ipadx*	space
int	*ipady*	interspace

By the fill type you specify how the component will use the free space. As before, the constants available are *NORTHWEST, NORTH, NORTHEAST, WEST, CENTER, EAST, SOUTHWEST, SOUTH* and *SOUTHEAST*, which are used as parameters for an object of type *GridBagConstraints*. In the program example you can see that the text box 3 is aligned using the *GridBagConstraints.VERTICAL* setting. This means that the component takes the maximum space in the vertical direction.

The developers of the GridBag layout have also thought about the space in between. Using the parameter *insets* you can specify the minimum external margin of the component. The values must be specified in pixel units (pixels). In contrast, the *ipadx* and *ipady* parameters specify the minimum internal margin. Again, this value must be specified in pixel units.

22.3.8 Swing

Since Java 2, another GUI library called Swing exists within Java SE in addition to the AWT. You can find most of the classes of this library in the package *javax.swing*. For all GUI components there are Swing classes that more than adequately replace the AWT classes (Figure 22.15).

Swing draws all interface elements itself and therefore does not provide a true native look-and-feel on all platforms, but tries to recreate it using the *UIManager* class found in the package *javax.swing*. This method has the disadvantage that the interface does not always look like native programs and – compared to native interfaces – is drawn relatively slowly. It has the advantage that the usual problems with porting graphical user interfaces are almost eliminated.

The problems arise from the different appearance and the different dimensions of similar surface components. This is further complicated by the fact that different operating systems have different fonts and font systems. As a result, a labeled button has a different dimension under Windows than under MacOS.

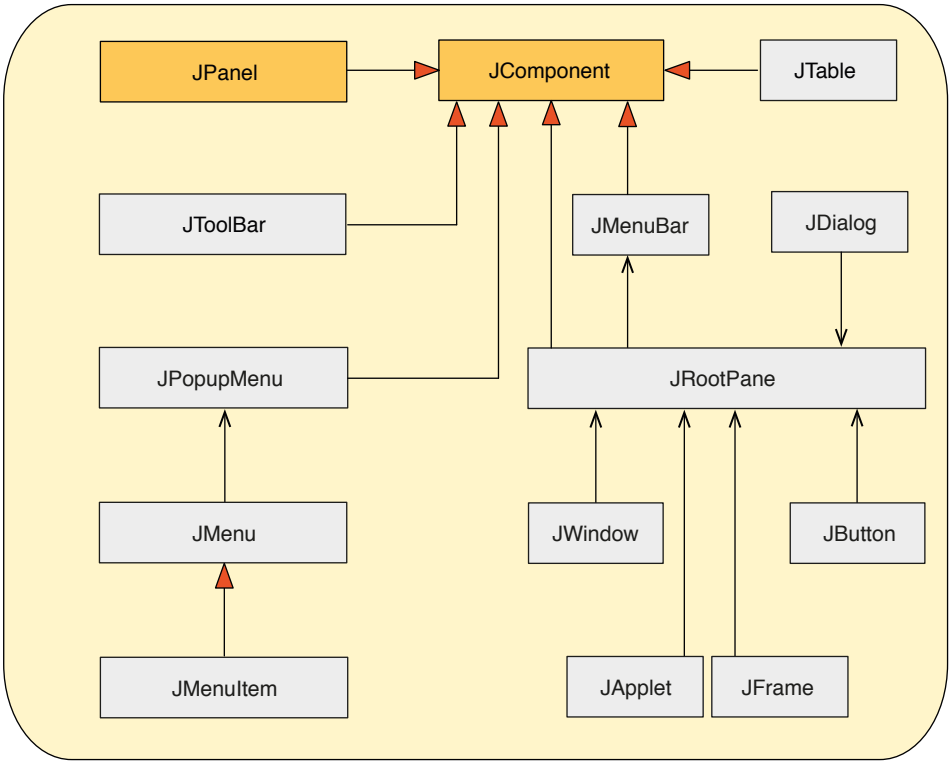


Figure 22.15: The main GUI classes of the Swing library.

Swing has been designed according to a variant of the »Model View Controller« (MVC) design pattern. This essentially means that content and the presentation of data are separated. All of Swing's UI elements are much more pleasant to program than their relatives from the AWT library. This is mainly because they have far fewer bugs and are more calculable when porting a program.

In Listing 22.21 you can see an example program for selecting different installed look-and-feels. In the first case, the "correct" look-and-feel is always set and the components are drawn as the user expects. In the second case, the program explicitly sets the Windows look-and-feel, in the third case the Metal look-and-feel, and in the last case the Motif look-and-feel.

Listing 22.21: Example of selecting a look-and-feel.

```
1 //Examples/Class_Libraries
2
3 package programmingcourse;
4
5 //Import AWT classes:
6 import java.awt.BorderLayout;
7 import java.awt.GridBagConstraints;
8 import java.awt.GridBagLayout;
```

```
import java.awt.Insets;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.UIManager;

public class LookAndFeelDemo extends JFrame {
  private static final long serialVersionUID = 1L;
  JPanel panel = new JPanel();
  JTextField textField1 = new JTextField();
  JTextField textField2 = new JTextField();
  JTextField textField3 = new JTextField();
  JTextField textField4 = new JTextField();
  JTextField textField5 = new JTextField();
  GridBagLayout gridBagLayout = new GridBagLayout();

  public LookAndFeelDemo() {
    try {
      init();
    } catch (Exception fehler) {
      fehler.printStackTrace();
    }
  }

  private void init() throws Exception {
    setSize(200, 150);
    this.setBackground(java.awt.Color.lightGray);
    textField1.setText("Text 1");
    textField2.setText("Text 2");
    textField3.setText("Text 3");
    textField4.setText("Text 4");
    textField5.setText("Text 5");
    panel.setLayout(gridBagLayout);
    getContentPane().add(panel, BorderLayout.CENTER);
    panel.add(textField1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
      GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(0, 0, 0, 0), 0, 0));
    panel.add(textField2, new GridBagConstraints(1, 0, 1, 1, 0.0, 0.0,
      GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(0, 0, 0, 0), 0, 0));
    panel.add(textField3, new GridBagConstraints(2, 0, 1, 2, 0.0, 0.0,
      GridBagConstraints.CENTER, GridBagConstraints.VERTICAL,
        new Insets(0, 0, 0, 0), 0, 0));
    panel.add(textField4, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0,
      GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(0, 0, 0, 0), 0, 0));
    panel.add(textField5, new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0,
      GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(0, 0, 0, 0), 0, 0));
  }

  public static void main(String[] arguments) {
```

```
61      try {
62        // 1. Systemunabhaengig: Natives Look + Feel
63        UIManager
64            .setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
65        // 2. Windows look + feel
66        //UIManager.setLookAndFeel(
67        //    "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
68        // 3. Metal look + feel
69        //UIManager
70        //    .setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
71        // 4. Motif look + feel
72        //UIManager.setLookAndFeel(
73        //    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
74      } catch (Exception e) {
75        e.printStackTrace();
76      }
77      LookAndFeelDemo fenster = new LookAndFeelDemo();
78      fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
79      fenster.setLocationRelativeTo(null); // Center the window
80      fenster.setVisible(true);
81    }
82
83  }
```

You should avoid setting the look-and-feel explicitly because not every virtual machine has all look-and-feels for licensing reasons. For example, the Aqua look-and-feel is only available on macOS. Exceptions are the Metal and Motif look-and-feels, which are available on every platform but rarely meet user expectations.

Figure 22.16 shows the different possible program outputs under the various operating systems supported by Java. In addition to the special look-and-feel called »metal«, a program can also be started to use the look-and-feel called native. Here, native means that Swing tries to reproduce the appearance of the original operating system interface as faithfully as possible. In most cases this works very well.

22.3.9 JavaBeans

JavaBeans are Java components with standardized interfaces and a simple programming model. There are two types of these components: visual and non-visual JavaBeans. Visual components are used to design graphical interfaces, while non-visual JavaBeans could be, for example, classes that facilitate access to databases.

Each JavaBean is characterized by the following features:

- Can be read by special tools.
- Can be customized by development tools
- Have event control capability
- Have attributes for customization
- Can be serialized (stored)

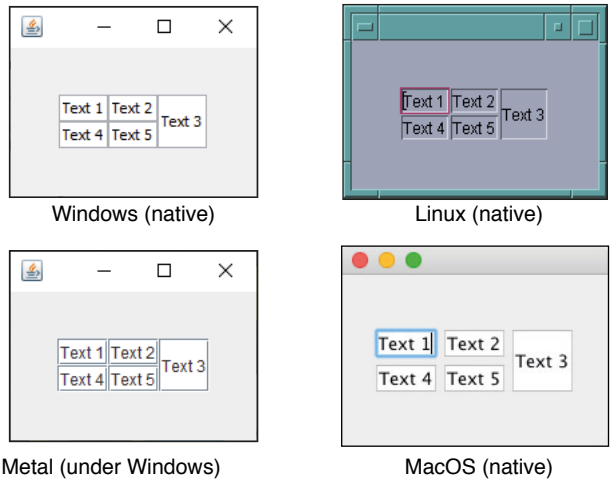


Figure 22.16: Swing supports various look-and-feels.

You have learned about JavaBeans in previous chapters, without this book saying a word about them. They are Java classes with getter and setter methods. If graphical components have such interfaces, they can be read and displayed very easily by development tools.

22.3.10 Applets

Applets are small Java programs that can be embedded in web pages and executed by a browser. These mini-programs are no longer state of the art. Oracle has discontinued this technology for this reason and has discontinued support for newer versions of Java. In other words, you should definitely not use applets anymore. There are far better technologies available, for example with HTML 5.

22.3.11 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) provides a unified interface to relational databases. In general, relational databases can be programmed via SQL. By means of the database language SQL information can be inserted, changed and deleted in relational databases. The Java Database Connectivity provides the programmer with various classes for this purpose, which are located in the package *java.sql*. Figure 22.17 shows the most important classes.

The advantage of JDBC is that it frees the developer from programming for a specific database, such as DB2, Oracle and HSQLDB. You can program against a standard interface and not have to worry about specialties of each relational database.

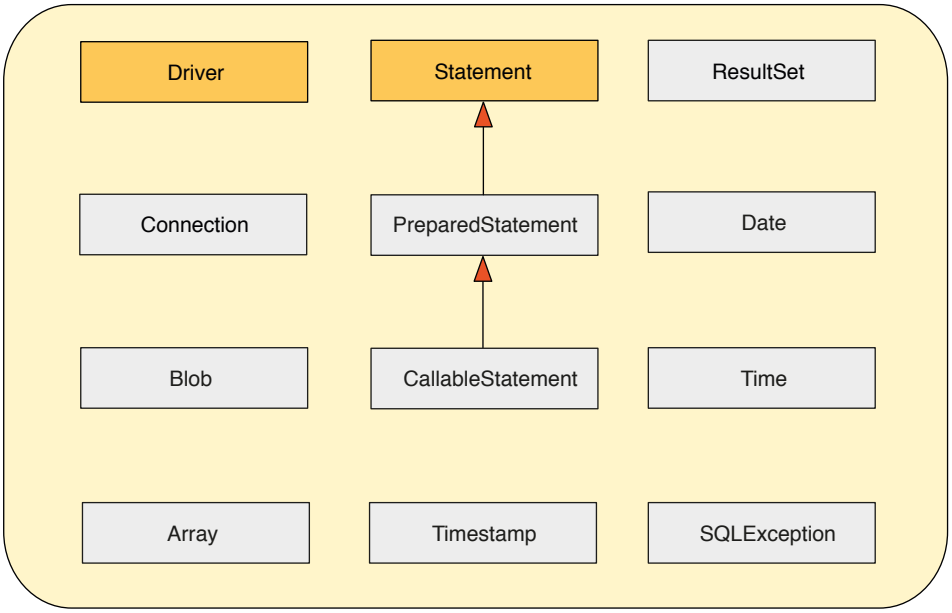


Figure 22.17: The main classes of the SQL package.

JDBC driver

The actual connection to the database is provided by a JDBC driver, which is usually developed by the database vendor. It decouples the Java program from the database. There are four classes of JDBC drivers:

- type 1: JDBC-ODBC bridge: This is the slowest driver type. It should only be used when relational databases that do not have an SQL interface need to be used.
- Native API driver: This driver type converts the JDBC calls to native API calls of the database. Therefore, you can communicate directly with the database.
- Net driver: As the name suggests, this driver type converts JDBC calls into database-neutral network calls. An intermediate layer between the driver and the database performs the final conversion.
- Thin drivers: This type of driver is mostly supplied by the database vendors themselves and is built directly on top of the database API. It is highly advisable to deal with these types in detail during database programming, because the quality of the DB connection stands and falls with the drivers. Accordingly, some drivers are expensive.

Which driver should you use? For the reasons mentioned above, driver types 3 and 4 are preferable.

22.3.12 Java Native Interface

The Java Native Interface (JNI) is a programming interface that allows you to use libraries that are native libraries from Java. Native libraries consist of machine code developed specifically for a particular operating system and processor. The JNI has three tasks:

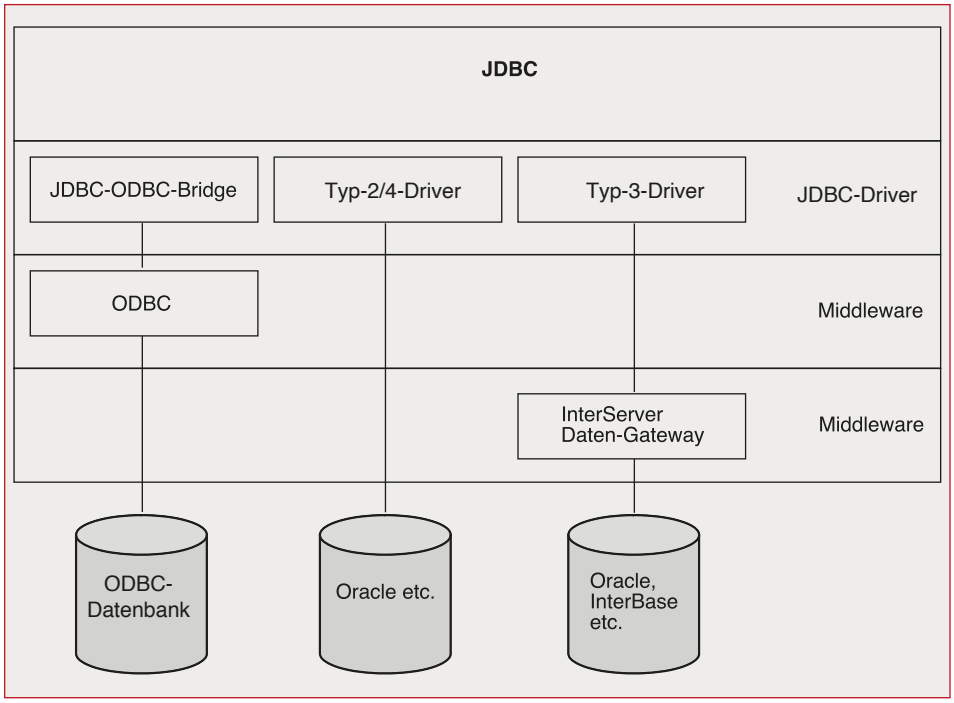


Figure 22.18: The different types of JDBC drivers.

- Access to operating system specific properties.
- Access to existing libraries
- Speed optimization

Especially in the early days of Java, when programs were not yet that performant, it seemed appealing to implement parts that were critical in terms of execution time in C or C++. Today, it is more interesting to use special libraries via the JNI that are only available as native libraries. Note that a Java program loses its portability with a single JNI call.

22.3.13 Remote Method Invocation

Remote Method Invocation (RMI) is an architecture for accessing methods of remote objects (remote methods). It can be used to write distributed Java applications very easily. This is because, unlike competing architectures such as CORBA, no additional product is required.

The RMI architecture specifies four access layers:

- Application layer: This layer represents client as well as server applications and contains the code to be executed.
- Remote layer: The remote layer could also be called the intermediary layer between the proxy layer and the underlying transport layer.

- Proxy layer: A proxy is a representative. Thus, the proxy layer forms the local proxy of objects that are executed on a remote computer. This is done via so-called stubs (hulls) and skeletons (skeletons), which are created with a special compiler.
- Transport layer: This layer probably needs no more large explanations. It takes care of the actual transport and is the counterpart to a socket communication.

22.4 Java Enterprise Edition

Unlike Java SE, Java Enterprise Edition (J2EE or JEE) is a collection of specifications. For these specifications, there are vendors such as the Apache Software Foundation (ASF) or the Eclipse community that provide class libraries based on the JEE specifications. The JEE addresses itself to larger enterprises, which would like to develop distributed applications. Distributed applications are programs that run not only on one computer, but on several. So that the objects of different computers can communicate with each other, there are different techniques. One technique is based on servlets and JavaServer Pages, another on the Enterprise JavaBeans component architecture. The Common Object Request Broker Architecture (CORBA) is another way to develop distributed applications.

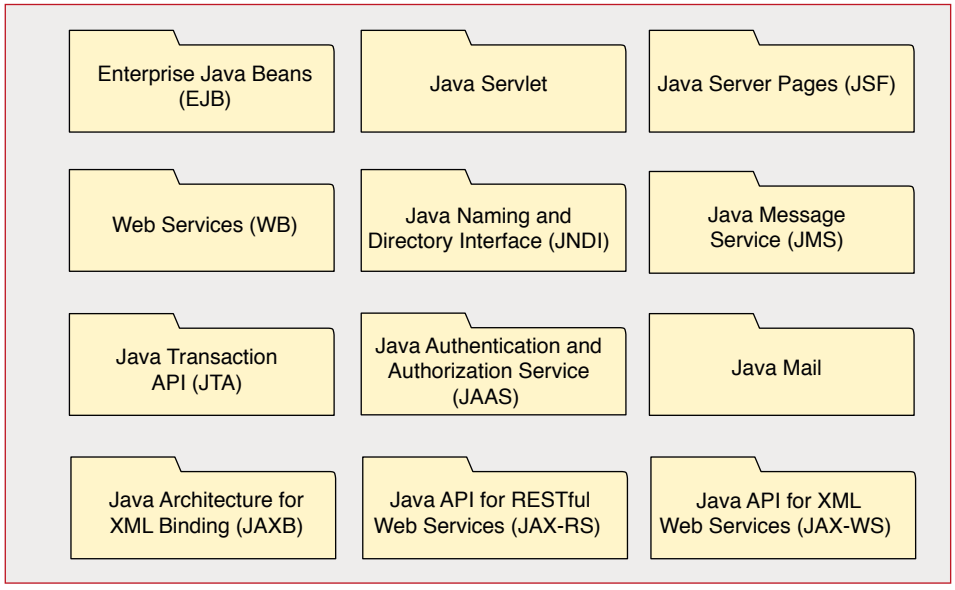


Figure 22.19: The main specifications of Java Enterprise Edition.

Enterprise JavaBeans (EJB) are like JavaBeans components. However, the programming model is much more powerful, but also more difficult to understand. The model is a widely evolved architecture for encapsulating enterprise data and business logic in standardized components. These standard components require an application server with an integrated EJB container as a runtime environment. The JRE as runtime environment is not sufficient. The EJB component model is structured as follows. There are three types of beans:

- Entity Beans
- Session Beans
- Message Driven Beans

The entity beans are primarily data carriers and encapsulate the business data, again distinguishing between two types:

- Entity Beans with Container-managed Persistence (CMP)
- Entity Beans with Bean-managed Persistence (BMP)

22.4.1 Entity Beans

An example of entity beans could be a person from a human resources database. However, entity beans can also be stored in simple files or relational databases. The location is not relevant to the Java programmer.

22.4.2 Session Beans

Session beans are the representatives of business logic. They provide a service and cannot be stored. There are two types:

- Stateless Session Beans
- Stateful Session Beans

Stateless session beans are very short-lived, stateless services that do not store any data from one method call to the next. They all have the same identity, and there is no way to distinguish them.

Stateful session beans, on the other hand, have state and store their data between multiple method calls. However, they are not persistent. After the server is shut down, their state is lost unless it is explicitly made persistent.

22.4.3 Message Driven Beans

Message Driven Beans are intended to support asynchronous communication, for example with message brokers. If one does not want to couple its application too strongly with another system, this kind of a this kind of an EJB is suitable.

22.4.4 Interfaces

Enterprise JavaBeans have two interfaces through which they can be accessed:

- Remote Interface
- Home Interface

The Remote Interface is used to access the actual component interface, for example to get the name of a person, while the Home Interface is used to affect the lifecycle of a bean. The Enterprise JavaBean architecture is a book-filling topic that I won't go into here.

22.5 Java Micro Edition

Java Micro Edition is intended for Java applications that are to run in mobile devices and so-called embedded systems. It is aimed at programmers of mobile devices and embedded systems and confuses the beginner with an almost even larger number of abbreviations than is the case in the other two editions.

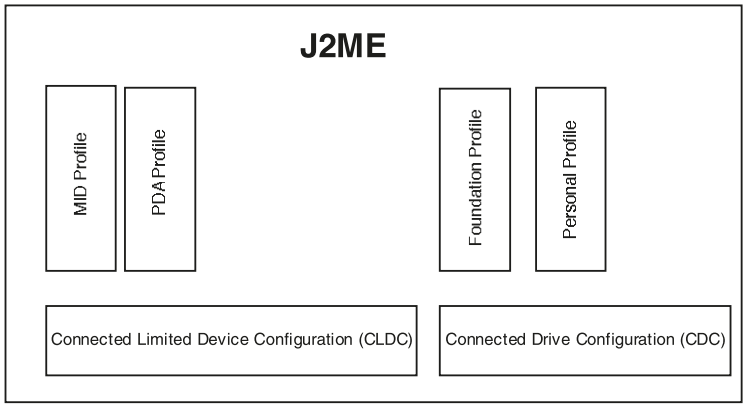


Figure 22.20: Java Micro Edition.

The edition consists of two parts: the Connected Limited Device Configuration (CLDC) and the Connected Drive Configuration (CDC). Programs that use the CLDC are executed by a special virtual machine, the Kilobyte Virtual Machine (KVM)). It is very small, very limited, and does not have, for example, decimal points. CDC programs, on the other hand, are executed by a complete but compact virtual machine (CVM).

Two profiles are based on the Connected Limited Device Configuration: the MID Profile and the PDA Profile. MID stands for Mobile Information Device, which refers to mobile devices. Analogous to servlets and applets, there is a special type of application for these devices, the MIDlets. These are delivered in archives and developed with various class libraries. These libraries are located in the following packages:

- *javax.microedition.io*
- *javax.microedition.lcdui*
- *javax.microedition.midlet*
- *javax.microedition.rms*

In addition, the basic libraries *java.io*, *java.lang* and *java.util* are also available, but they do not provide floating point types due to the limited virtual machine. Java programs designed using Java SE or even JEE are not portable to mobile devices. Because of the many limitations, Java Micro Edition has little relevance today.

22.6 External Class Libraries

Outside of the Java editions, there is an unmanageable amount of other class libraries. The three most important sources are the Apache Software Foundation, the Eclipse Community and SourceForge.

22.6.1 Apache Software Foundation

The Apache Software Foundation (ASF) contains the largest collection of freely available libraries for Java programming. Freely available means that not only is the software open source, but you may also use the software without paying a fee. The Apache license is designed so that this also applies to commercial software. You will find in the ASF mainly solutions for professional Java programs. The Apache libraries are therefore used by many companies that develop Java software. You can find more details on the ASF homepage (*https://www.apache.org*).

22.6.2 Eclipse Community

The Eclipse Community is a creation of IBM. It all started with the development of the open source Eclipse development environment (Eclipse IDE). From this, IBM developed the open-source Eclipse Rich Client Platform (Eclipse RCP). The Eclipse Community was founded in 2004 to further develop this open-source platform. Today, you can not only obtain the Eclipse IDE and the associated platform from the Eclipse Community, but also many other open source and free Java class libraries. As with the ASF, the software of the Eclipse Community can be used free of charge, even for commercial applications. For more information, please visit the Eclipse Community homepage (*https://www.eclipse.org*).

22.6.3 SourceForge

SourceForge is not a non-profit organization like ASF or the Eclipse Community, but privately owned. The web portal is used by many free software developers to distribute their open-source products. Here you can find Java class libraries, which are not tailored to professional programs. For more information, please visit the SourceForge homepage (*https://www.sourceforge.net*).

22.6.4 Other Open-Source Software

The Java-Source website provides a good overview of many open-source Java class libraries. For more details, see the Java Source homepage (*https://java-source.net/*).

22.6.5 Commercial Software

Besides the well-known providers of open source Java class libraries, there are many commercial industry solutions. Almost all major software houses like SAP, Software AG, RedHat, Amazon or Tibco have developed commercial Java libraries which they distribute with their products.

22.7 Summary

Java has a small language core that is extended by so-called class libraries. Java class libraries are nothing more than collections of related Java classes. Java class libraries can be divided into the official Java class libraries, open source class libraries of developer communities and private providers as well as commercial class libraries. The official Java class libraries are part of Java and are organized into three editions: Java Standard Edition (JSE), Java Enterprise Edition (JEE) and Java Micro Edition (JME).

The most important of the three editions is the Java Standard Edition. It corresponds approximately to the Java Development Kit (JDK). In this edition, not only the JDK tools and the Java Runtime Environment (JRE) are contained, but several thousand useful classes for graphic surfaces, file input and output, strings, event and error handling as well as components.

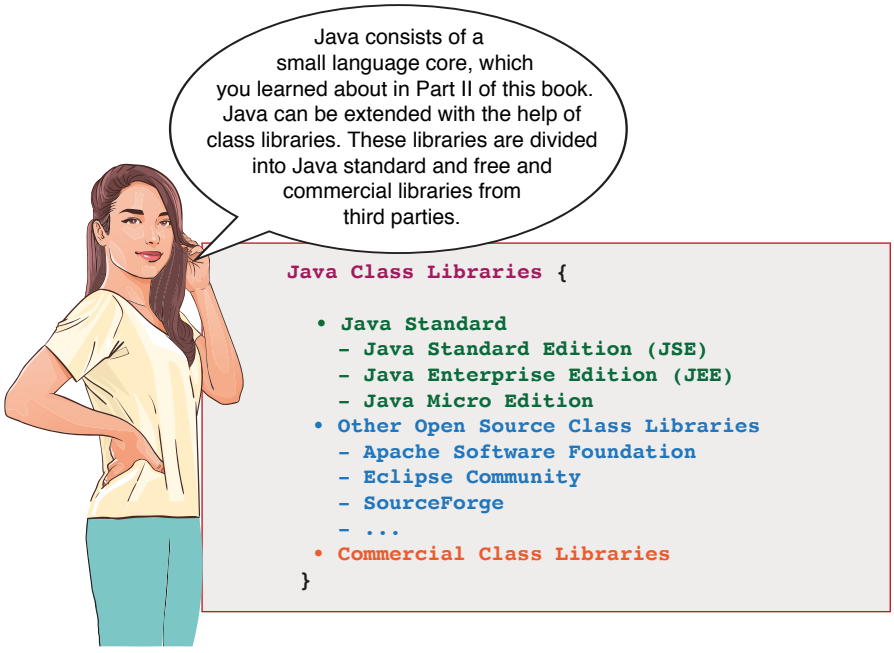


Figure 22.21: Julia thinks it's good that there are so many freely available class libraries.

Java Enterprise Edition is primarily a collection of specifications. These specifications implemented various companies and organizations in the form of Java class libraries. The class libraries are aimed primarily at professional software developers who want to develop distributed applications. The programming of small devices supports the Java Micro edition. This edition has today hardly more meaning.

22.8 Literature

Java API Specification: *https://docs.oracle.com/javase/10/docs/api/overview-summary.html*

22.9 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/class_libraries
There you will also find the solutions to the exercises.

23 Rules

23.1 Introduction

As with any programming language, Java programs have certain rules. Some rules are soft – they have no negative effect on the execution of the program. Other rules lead to the fact that the program cannot be compiled. Still other rules have the consequence that the program seems to work – but unfortunately completely different than you imagine.

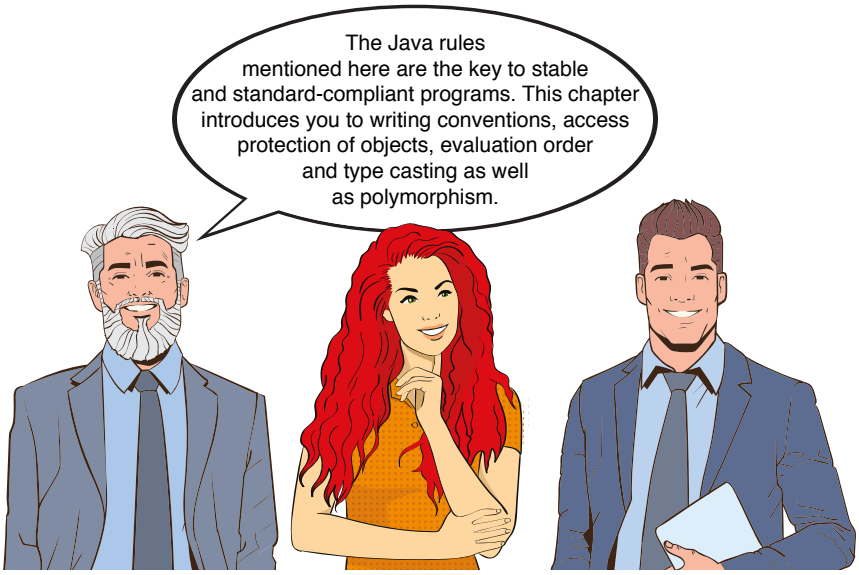


Figure 23.1: The Roth family relationships provide illustrative examples of Java rules.

In other words, it can't hurt to learn a little more about the rules of Java programming. This chapter introduces you to all the rules, knowledge of which will ensure that your programs work as they should.

23.2 Overview

When developing Java programs, you must respect many rules. This starts with notations of packages, classes, interface, methods, variables and constants and continues with the rules for visibility of classes, methods and variables. In addition there are rules concerning the evaluation order, rules for the type conversion, and rules for polymorphism.

Writing conventions are soft rules. They are only necessary, so that one can read programs better and have no effect on the execution of programs. The situation is different with access levels: The keywords *public, protected, private* and default access level have a significant impact on the design of programs.

Table 23.1: Overview of the Java rules in this chapter.

Rule	Description	Purpose
Writing conventions	The writing conventions define how language elements (packages, classes, interfaces, methods, variables and constants) should be written within the Java source code.	Better readability and uniform notation of the Java source code
Access levels	Access protection determines who may access objects, methods and attributes within Java programs.	Protection against unwanted access to objects and their elements
Evaluation order	The evaluation order determines the order in which expressions are processed within Java programs.	Control of the evaluation order of Java instructions
Type casting	Type casting temporarily removes the assignment of data types to variables.	Controlled change of the assignment of variables to data types
Polymorphism	Polymorphism allows methods to be overwritten and overloaded.	Situational change of the behaviour of objects

Another stumbling block when developing Java programs is the evaluation order of expressions. Equally unpleasant are confusions in implicit and explicit type conversion. Last but not least, the rules for polymorphism, overloading and overwriting methods are extremely important programming rules.

Eclipse can only keep track of a portion of these many rules without extensions. If you want more support from the development environment, take a look at my homepage. There are some useful Eclipse extensions listed, whose explanation would have gone beyond the scope of this book (Section 23.9, »Literature«).

23.3 Writing Conventions

Let's start with the simplest rules in programming, the writing conventions. There are a number of recommendations for writing identifiers in Java programs. The reason for these writing conventions was to make Java programs more uniform and to increase the readability of the source code. Table 23.2 summarizes the internationally recognized writing conventions in Java programs.

Table 23.2: This table presents an overview of the Java writing conventions.

Element	Writing convention	Example
Packages	Generally, packages are written in lowercase letters. The idea was to build them like Internet addresses. The basic package is a top-level domain, followed by the company name, then the project name and so on.	net.steppan.transfer; net.programmingcourse.statistic
Classes	Classes are written like nouns. The first letter is always capitalized. If the noun is written coupled, a so-called camel-back spelling is recommended, in which each word beginning is introduced with a capital letter. In any case, the spelling must be exactly like the file name of the class.	class Student; class StatisticApp;
Interfaces	Like classes	interface Creature;
Methods	Methods are written like verbs with a leading lower case letter. In some cases, a camel-back notation is also useful here, for example with getters and setters. Here, one uses a prefix followed by the attribute name.	setName(); isStudent();
Variables	The first letter of variables is written in lower case; the camel-back notation can also be used here.	String firstName; String csvFile;
Constants	In general, constants are written in capital letters. Identifiers that consist of several words are separated with an underscore.	public static final int COUNT_LEGS = 2;

The names of identifiers should not contain special national characters and also must not contain spaces. In addition, it is not allowed to use special characters for class names and identifiers that are reserved for other Java language elements.

23.4 Access Protection

23.4.1 Four Levels of Access Protection

Java allows objects, methods and attributes to be protected. This is done either via access levels or via the rules for the scope of validity of identifiers. You may define two different access levels for classes: *public* and *default*. It is important with class definitions that only one of the classes in a class file may be *public* and thus may be publicly accessible. For methods and attributes, the Java programmer even has four access levels available (Table 23.3).

Table 23.3: Java has four access levels.

Keyword	Access level	Impact
private	Class	Only visible within the *current* class.
default	Package	Only visible within the package.
protected	Inheritance & Package	Only visible to child classes and within the package.
public	Public	Visible in the whole program.

The access levels *private*, *protected* and *public* are defined by the keyword of the same name. If you want to restrict access to the *default* area, simply omit the leading keyword for the access area. The second highest level of protection then comes into effect. Figure 23.2 shows the effects of the access scopes using a class diagram.

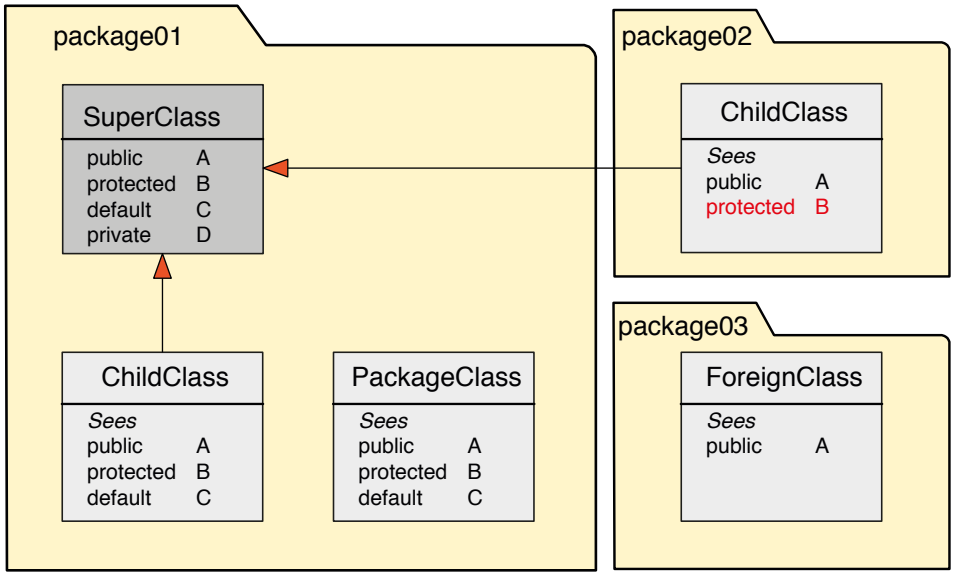


Figure 23.2: Access levels of Java classes.

The diagram shows five classes in three packages. Two classes named *ChildClass* are derived from a *SuperClass*. Superclass is another name for a base class. One of the two child

classes is inside the package containing the *SuperClass*. The other *ChildClass* is outside. Inside the package there is also a class named *PackageClass* for this reason. Outside the package is a class called *ForeignClass*.

23.4.2 Access Level »private«

Using the *private* keyword guarantees the strongest protection against access from other objects. Only objects of the same class may access private methods and variables. Classes, of course, cannot be provided with this keyword for obvious reasons, otherwise the class would be untraceable.

23.4.3 Access Level »default«

The second highest encapsulation strength is the default section. Only classes that are in the same package have access to identifiers in the *default* section. In Figure 23.2 you can see that the classes *ChildClass* and the *PackageClass* get access to the methods of the base class.

23.4.4 Access Level »protected«

The next weaker level of security is achieved by using the *protected* keyword. In Figure 23.2 you can see that this confidential area can only be accessed by objects of a class that is either derived or in the same package. Elements of this class can only be accessed by objects that are within the same package or are derived.

23.4.5 Access Level »public«

The *public* keyword provides the weakest form of access protection. All information is public. You can see from Figure 23.2 that all classes in the diagram can access method *A* of the *base class* defined as *public*.

The *A* method declared as *public* is accessible to all other classes, even outside the package. The protected method *B*, on the other hand, can only be used by the two child classes and the foreign class. The method *C* (default) is even visible only to the child class inside the package and the foreign class, while the private method belongs only to itself. So much for the general rules.

23.4.6 Case Study

To implement access protection in a Java program, let's take the private environment of Professor Karsten Roth as a case study. He is married to his wife Sophie for the second time. His daughter Katharina and his son Max are from his first marriage. Katharina lives

with her father and bears his last name »Roth«. Max, on the other hand, lives with his mother and bears her last name »Stern«. In addition, Professor Roth has a number of students in his semester at his university, for example Anna, whom you already know from the programming course.

Based on this information, a class diagram can be drawn as a template for a Java program (Figure 23.3). The diagram shows the implementation with exactly the same packages as before. The classes, however, already have names specific to the case study.

Note that the class *Child* exists twice. The similarity of names is not an error or contradiction. Java allows classes with the same name as long as they are in different packages. Why becomes clear if we write out the name of a class fully qualified, that is, with its package name: The class *Child* is called *roth.Child* on the left side of the figure with the full class name and *stern.Child* on the right side.

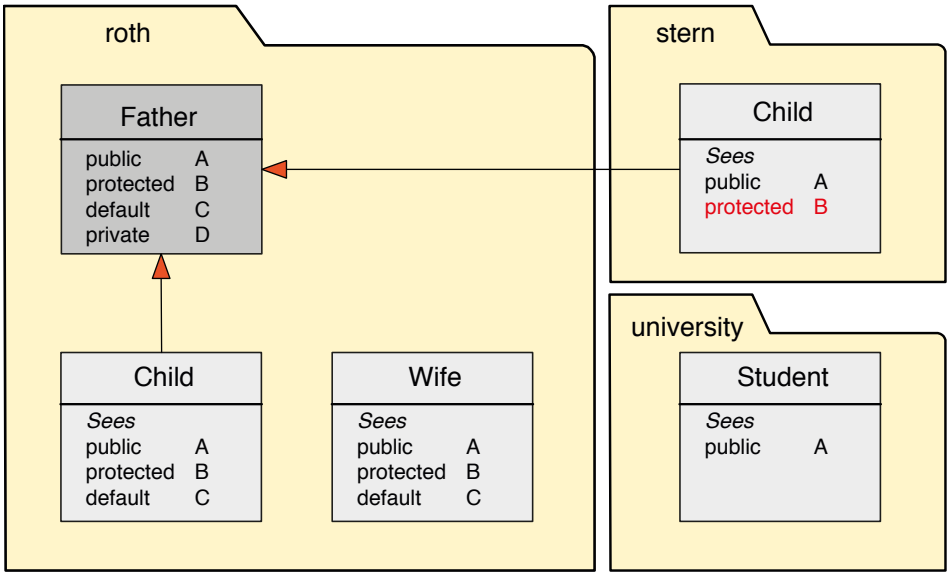


Figure 23.3: Access protection with the example of a class diagram.

From this class diagram we derive an object diagram in the next step. The object *karsten* (Professor Roth) is created from the super class *Father* and the child object *katharina* from the class *Child* of the package *roth*. The child object *max* arises from the class *Child* of the package *stern*. The wife object *sophie* is created from the class *Wife*. The student object *anna* arises from the class *Student*. You can see the corresponding scenario in the form of an object diagram in Figure 23.4.

The next step is to implement the classes and objects in Java. In Listing 23.1 you can see the super class *Father*. This class has four methods that correspond to the four access areas of Java: *getPublicArea()*, *getProtectedArea()*, *getDefaultArea()*, and *getPrivateArea()*. In addition, the class still has a method *getArea()*, which is publicly accessible and is a kind of backdoor. You will see in a moment what these methods are all about.

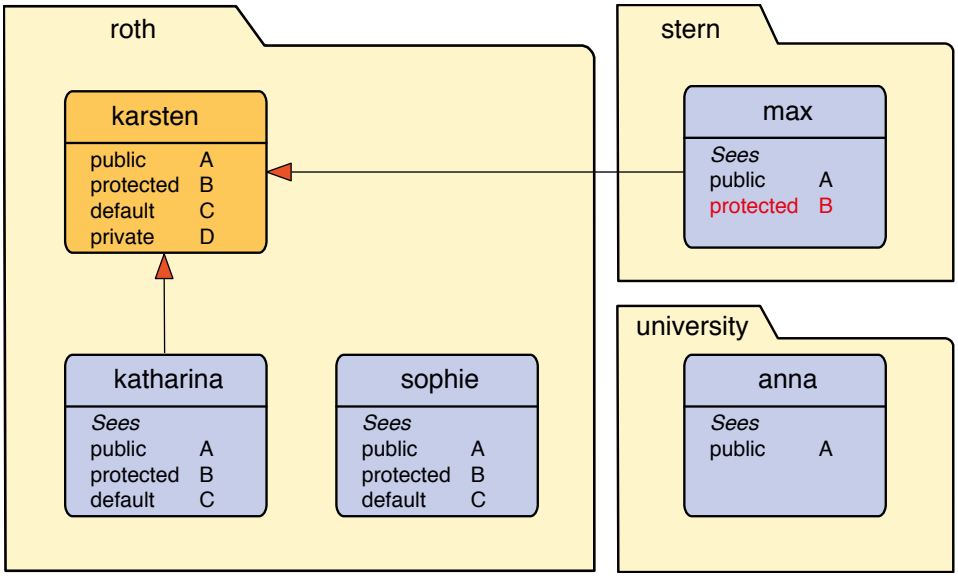


Figure 23.4: Access protection with the example of an object diagram.

Listing 23.1: The class »Father« is the template for the object »karsten«.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7
8    public String getPublicArea() {
9      return "Public";
10   }
11
12   protected String getProtectedArea() {
13     return "Protected";
14   }
15
16   String getDefaultArea() {
17     return "Default";
18   }
19
20   private String getPrivateArea() {
21     return "Private";
22   }
23
24   public String getArea() {
25     return getPrivateArea()  + " / " +
26            getDefaultArea() + " / " +
27            getProtectedArea() + " / " +
```

```
28            getPublicArea();
29    }
30  }
```

Next, we implement the *child* class from the *roth* package, the template for the *katharina* object.

Listing 23.2: The class »Child« is template for the object »katharina«.

```
1  //Examples/Rules
2
3  package roth;
4
5  /**
6   * Class for child objects like Katharina
7   */
8  public class Child extends Father {
9
10   public String getArea() {
11     return super.getDefaultArea() + " / " +
12            super.getProtectedArea() + " / " +
13            super.getPublicArea();
14   }
15 }
```

As you can see from the Listing, the daughter Katharina has the most favorable prerequisites of all participants: Since she is a direct descendant of the father, she only has to use the keyword *super* to access the protected methods of the superclass *Father* and thus of the object *karsten*. The keyword *super* is reserved for accessing the superclass (base class).

The wife has it similarly good. To access her husband's protected and default methods, she only has to create a new object of the father (from her husband's point of view) (Listing 23.3).

Listing 23.3: The class »Wife« is the template for the object »sophie«.

```
1  //Examples/Rules
2
3  package roth;
4
5  /**
6   * Template for wife objects like Sophie
7   */
8  public class Wife {
9
10     public String getBereich() {
11       Father carsten = new Father();
12
13         return carsten.getDefaultArea() + " / " +
14                carsten.getProtectedArea() + " / " +
15                carsten.getPublicArea();
16     }
```

```
17 }
```

From this example with the wife, you can see how important it is, even in Java, to check carefully with whom you have intimate relationships. In Java, classes of the same package enjoy almost complete access to the entire information of the package's roommates. In contrast, things look comparatively bad for Professor Roth's son. He does not even get the same access as Karsten Roth's wife, since he is not in the father's package (Listing 23.4).

Listing 23.4: The son has access to the protected area of the super class through »super«.

```
1  //Examples/Rules
2
3  package stern;
4
5  import roth.Father;
6
7  /**
8   * TEmplate for child objects like Max
9   */
10 public class Child extends Father {
11
12   public String getArea() {
13       return super.getProtectedArea() + " / " +
14              super.getPublicArea();
15
16       //Not allowed:
17       //return new Father().getProtectedBereich();
18   }
19 }
```

The son cannot access the default area because he comes from a foreign household (package). However, he at least has access to the protected area of the super class *Father*, which he can see via the keyword *super*. So he does not have to create a new object of the father, as it was the case with the wife. However, if he is not skilled at accessing it, things look even worse for him. Subsequent access via a new object of the super class *Father* would be denied to him despite the inheritance relationship:

```
new Father().getProtectedBereich();
```

Compared to the son, the student Anna has only very limited access to the class *Father* and thus to the object *karsten*. She is neither related to Professor Roth nor does she live with him under the same roof (package). Bad luck for her, because she is treated like a stranger and only gets access to the public area (Listing 23.5).

Listing 23.5: The student Anna has access only to the public area.

```
1  //Examples/Rules
2
3  package university;
4
5  import roth.Father;
```

```
6
7  /**
8   * Template for child objects like Anna
9   */
10 public class Student {
11
12   public String getArea() {
13     return new Father().getPublicArea();
14   }
15 }
```

Finally, the following main program summarizes all the relationships of Professor Roth, his wife, the two children, and the student Anna (Listing 23.6).

Listing 23.6: The summary of accessibility options.

```
1  //Examples/Rules
2
3  package roth;
4
5  import university.Student;
6
7  public class Family {
8
9    public static void main(String[] args) {
10
11     // Create objects:
12     Father karsten = new Father();
13     Wife sophie = new Wife();
14     Child katharina = new Child();
15     stern.Child max = new stern.Child();
16     Student anna = new Student();
17     // Show the access areas :
18     System.out.println("\nAccess to the areas of the class Father:\n");
19     System.out.println("Karsten (Father): "
20       + karsten.getArea() + "\n");
21     System.out.println("Katharina (Daughter): "
22       + katharina.getArea() + "\n");
23     System.out.println("Sophie (Wife): "
24       + sophie.getBereich() + "\n");
25     System.out.println("Max (Son): "
26       + max.getArea() + "\n");
27     System.out.println("Anna (Student): "
28       + anna.getArea() + "\n");
29   }
30 }
```

There are a couple of special features in this sample program. The first special feature is that the program imports the two child classes with their full, that is, fully qualified, names. The fully qualified name is necessary when using classes with the same name from different packages. In this case it is needed so that the compiler can distinguish the child from

the package *roth* and the other from the package *stern*. One child should eventually become the daughter object, the other the son.

The second peculiarity is that all classes have a public method called *getArea()*. In this method, all classes call the methods of the class *father*, to which they have maximum access. Consequently, the *father* class itself also has this method. It uses this method to access its own private method *getArea()*. Only it is allowed to do this. The tricky visibility ranges can be seen in the program output:

```
Access to the areas of the class father:
Karsten (father): Private / Default / Protected / Public
Katharina (daughter): Default / Protected / Public
Sophie (wife): Default / Protected / Public
Max (son): Protected / Public
Anna (student): Public
```

23.4.7 Scope of Variables

Now it would be nice if we could use Roth family relationships to check off the visibility of classes, methods, and variables. But that's not the case, because Java knows four different types of variables. Some of them can be hidden by other variables – and this without applying encapsulation (Table 23.4).

Table 23.4: Variable shadowing.

Variable	Shadowing
Class variables	Possible
Object variables	Possible
Parameters	Impossible
Local variables	Impossible

What is meant by the shadowing of variables? It means that a variable with the same name overlaps another variable in a block. To show this, we will use the class *father* again as an example. This time it gets as attribute an object variable named *privateArea*. This variable is valid for the entire object, since it was not declared in any local block, but for the entire class and thus also for *every* object. This object variable is immediately assigned *private* (Listing 23.7). Because of this assignment, when the *getPrivateArea()* method is accessed, the class returns *Private* as expected.

Listing 23.7: The class »Father« with instance variable and method.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    private String privateArea = "Private";
8
9    public String getPublicArea() {
10     return "Public";
11   }
12
13   protected String getProtectedArea() {
14     return "Protected";
15   }
16
17   String getDefaultArea() {
18     return "Default";
19   }
20
21   private String getPrivateArea() {
22     return privateArea;
23   }
24
25   public String getArea() {
26     return getPrivateArea()  + " / " +
27            getDefaultArea() + " / " +
28            getProtectedArea() + " / " +
29            getPublicArea();
30   }
31 }
```

The situation is different if you introduce a local variable called *privateArea* and assign it the value *Local* in line 22 of the Listing 23.8. In this case, when you call the *getPrivateArea()* method, you get back the value *Local*. This is because the local variable named *Private* has temporarily overlaid the object variable of the same name in the block of the *getPrivateArea()* method.

Listing 23.8: The class »Father« with object variable and method.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    private String privateArea = "Private";
8
9    public String getPublicArea() {
10     return "Public";
```

```
11    }
12
13    protected String getProtectedArea() {
14      return "Protected";
15    }
16
17    String getDefaultArea() {
18      return "Default";
19    }
20
21    private String getPrivateArea() {
22      String privateBereich = "Local";
23      return privateBereich;
24    }
25
26    public String getArea() {
27      return getPrivateArea()  + " / " +
28             getDefaultArea() + " / " +
29             getProtectedArea() + " / " +
30             getPublicArea();
31    }
32  }
```

Katharina is confused and wants to know what happened to the global object variable and when it will reappear.

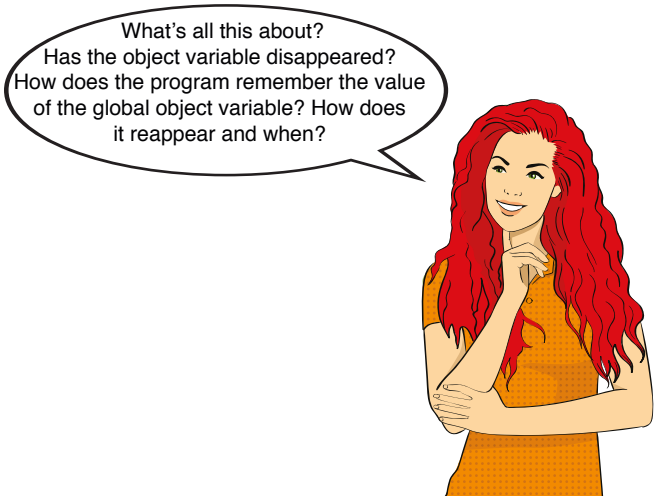


Figure 23.5: Katharina wants to know more details about shadowing of variables.

First: the object variable is not completely gone. Its *Identifier* has only been overshadowed, which is why no one needs to remember its value. The object variable and the local variable live side by side. It is no problem to use them in the same block. All you have to do is call them appropriately – how is shown in Listing 23.9.

Listing 23.9: variable and object variable in unison.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    private String privateBereich = "Private";
8
9    public String getPublicBereich() {
10     return "Public";
11   }
12
13   protected String getProtectedBereich() {
14     return "Protected";
15   }
16
17   String getDefaultBereich() {
18     return "Default";
19   }
20
21   private String getPrivateBereich() {
22     String privateBereich = "Local";
23     return this.privateBereich;
24   }
25
26   public String getBereich() {
27     return getPrivateBereich()  + " / " +
28            getDefaultBereich() + " / " +
29            getProtectedBereich() + " / " +
30            getPublicBereich();
31   }
32 }
```

The program uses the *this* keyword on line 23 to access the current object and thus the global variable *privateArea*. This puts the class back on track: the *getPrivateArea()* method returns *Private* as the area, as intended. The only unfortunate thing about the last example is that the local variable has become completely redundant. Eclipse marks the variable with the warning *The value of the local variable privateArea is not used.* In other words, the variable has no effect.

To change this, another example: the class *father* should have a method to set the value of the private area attribute. To do this, the method should access the object variable *privateArea* (Listing 23.10).

Listing 23.10: variable and object variable in harmony.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
```

```
6
7     private String privateBereich = "Private";
8
9     public String getPublicBereich() {
10      return "Public";
11    }
12
13    protected String getProtectedBereich() {
14      return "Protected";
15    }
16
17    String getDefaultBereich() {
18      return "Default";
19    }
20
21    private String getPrivateBereich() {
22      return privateBereich;
23    }
24
25    public void setPrivateBereich(String privateBereich) {
26      this.privateBereich = privateBereich;
27    }
28
29    public String getBereich() {
30      return getPrivateBereich()  + " / " +
31             getDefaultBereich() + " / " +
32             getProtectedBereich() + " / " +
33             getPublicBereich();
34    }
35  }
```

As you can see from the *setPrivateArea()* method in lines 25 to 27, the parameter (local variable) and object variable again coexist in harmony. Since there is no local variable in the getter method in lines 21 to 23, the *return privateArea* statement is again sufficient to access the global variable. When you start the program, the method outputs the following:

```
Access to the areas of the class father:
Karsten (father): Private / Default / Protected / Public
Katharina (daughter): Default / Protected / Public
Sophie (wife): Default / Protected / Public
Max (son): Protected / Public
Anna (student): Public
```

This concludes the excursion into the world of visibility rules.

23.5 Evaluation Order

For the evaluation order of a Java program, the rules known from part II, »Java Language«, apply. Actually, one could put the topic aside, if one believed that the evaluation sequence

runs exactly, as used from mathematics. The following two examples show that this is not always the case and that it is highly recommended to use parentheses.

23.5.1 Dot Before Dash

At the beginning of the calculation example (Listing 23.11), four variables are agreed upon as integer values (lines 8 to 11). Then the program uses them in a mixed addition and multiplication in five different cases.

Listing 23.11: Addition and multiplication mixed.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class DotBeforeDashDemo {
6
7    public static void main(String[] arguments) {
8      int a = 4;
9      int b = 4;
10     int c = 3;
11     int d = 4;
12     int result;
13     result = a + b * c + d;
14     System.out.println("Case 1:   4 +  4  *  3  + 4  = " + result);
15     result = a + (b * c) + d;
16     System.out.println("Case 2:   4 + (4  *  3) + d  = " + result);
17     result = (a + b) * c + d;
18     System.out.println("Case 3:  (4 +  4) *  3  + d  = " + result);
19     result = ((a + b) * c) + d;
20     System.out.println("Case 4: ((4 +  4) *  3) + 4  = " + result);
21     result = (a + b) * (c + d);
22     System.out.println("Case 5:  (4 +  4) * (3  + 4) = " + result);
23   }
24 }
```

The program outputs the following values:

```
Case 1: 4 + 4 * 3 + 4 = 20
Case 2: 4 + (4 * 3) + d = 20
Case 3: (4 + 4) * 3 + d = 28
Case 4: ((4 + 4) * 3) + 4 = 28
Case 5: (4 + 4) * (3 + 4) = 56
```

The explanation of the results does not seem difficult.

No Bracket Levels

In the first case, the program does not use parenthesis levels. For this reason, result 1 is identical to case 2, because first the program performs the multiplication, gets 12 as an

intermediate result, adds 4 twice, and finally gets 20. In the second case, this is only more obvious because of the bracket levels.

From The Inside Out

In the third and fourth cases, the program forces the left addition to be done before the multiplication. Thus, the expression is processed from the inside out. The rule that an expression must always be processed from left to right is overridden.

Dash Before Dot

In the fifth and last case, the two additions are processed first because they are parenthesized. Then the program multiplies them together. Because of the parentheses, the program achieves that dash before the dot applies and not the dot before the dash. This rule should be known from mathematics.

23.5.2 Dot Before Dot

The situation is somewhat different when different types of point calculations (*, /, %) are used together. Listing 23.12 shows an example of this.

Listing 23.12: Addition, division and multiplication mixed.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class DotBeforeDotDemo {
6
7    public static void main(String[] arguments) {
8      int a = 4;
9      int b = 4;
10     int c = 3;
11     int d = 4;
12     int result;
13     result = a + b * c / d;
14     System.out.println("Case 1:   4 +  4  *  3  / 4   = " + result);
15     result = a + (b * c) / d;
16     System.out.println("Case 2:   4 + (4  *  3) / 4   = " + result);
17     result = a + (c * b) / d;
18     System.out.println("Case 3:   4 + (4  *  3) / 4   = " + result);
19     result = a + b * (c / d);
20     System.out.println("Case 4:   4 +  4  * (3  / 4)  = " + result);
21     result = a + (b * (c / d));
22     System.out.println("Case 5:   4 + (4  * (3  / 4)) = "+ result);
23     result = (a + b) * c / d;
24     System.out.println("Case 6:  (4 +  4) *  3  / 4   = " + result);
25     result = ((a + b) * c) / d;
26     System.out.println("Case 7: ((4 +  4) *  3) / 4   = " + result);
```

```
27      result = (a + b) * (c / d);
28      System.out.println("Case 8:  (4 +  4) * (3  / 4)  = " + result);
29    }
30  }
```

As with the previous example, the spreadsheet program agrees on four variables as integer values. It then uses them in a mixed addition, division, and multiplication in eight cases. The result of this calculation in four different ways looks like this:

```
Case 1: 4 + 4 * 3 / 4 = 7
Case 2: 4 + (4 * 3) / 4 = 7
Case 3: 4 + (4 * 3) / 4 = 7
Case 4: 4 + 4 * (3 / 4) = 4
Case 5: 4 + (4 * (3 / 4)) = 4
Case 6: (4 + 4) * 3 / 4 = 6
Case 7: ((4 + 4) * 3) / 4 = 6
Case 8: (4 + 4) * (3 / 4) = 0
```

The result is again obvious at first glance. The evaluation order is from left to right, with the dot being preferred over the dash. It should be noted that the division sign is considered a dot (as in mathematics).

Dot Before Dash

In the first case, the program tries to perform addition first, but is not allowed to do so because of the »dot before dash« rule. For this reason, it first performs the multiplication and gets 12 as the result. It divides this value by *b* and gets 3. The final addition of 3 plus 4 gives 7 again.

Excess Bracket Levels

In the second case, the set parentheses do not change the result, since they do not affect the evaluation order. Thus, the calculation proceeds identically.

Swap

In the third case, a mathematical rule comes into effect, the so-called law of interchange (commutative law). This means that equivalent operands (arguments) can be interchanged. So whether *b * c* is processed first or *c * b,* does not matter at first for equivalent arithmetic operations. So the calculation runs identically here, too.

Lost Division Remainder

What happened in the 4th case? The bracketing of the expression *c* / *d* leads in contrast to the first case to the fact that here an intermediate result must be formed. This also happened in the fifth and eighth case, because the expression could not be multiplied immediately.

In the case of division, a type conversion takes place (Section 23.6, »Type Conversion«). Since the result 0.75 is not an integer value, the decimal places (division remainder) are lost. What remains is only the value 0, and as a multiplier in the seventh case, this also gives 0 as the result.

From The Inside To The Outside

In cases 6 and 7, the processing of the expression proceeds from the inside to the outside. First the addition is calculated, then the multiplication, and finally the division.

23.6 Type Conversion

Java, unlike some other languages, is strictly typed. If an identifier has been declared as a certain type, this is valid for the whole flow of the program. The compiler checks for assignments to see if the data types match. The following is not compiled:

```
long a = 1;
int b = 1;
b = a; // will not be compiled
```

The compiler checks if a data loss can occur. Since the variable *a* can occupy a larger range of values than *b*, it refuses to approve the expression. The following, on the other hand, succeeds:

```
long a = 1;
int b = 1;
a = b; // will be compiled
```

However, the fact that an identifier is chained to its type for life does not mean that it is not possible to break these shackles *temporarily*. Type conversion allows values to be exchanged between variables of different types, as the following fragment shows:

```
long a = 1;
int b = 1;
int c = 1;
b = (int)a; // will be compiled
c = a; // will not be compiled
```

As the example shows, the program performs a temporary conversion of *a* through the cast operator to pass the value to *b*. You can see that this arrangement is only temporary in the line that follows. Here the program tries again to pass a value of type *long* to a variable of type *int*, and fails again because the compiler rightly prevents it.

The Java programming language has two types of type conversion. Implicit type conversion is done automatically. Here the developer is passive. With the explicit conversion, you explicitly perform a conversion.

23.6.1 Implicit Conversion

The implicit conversion is tricky, where a creeping value conversion takes place. The compiler does not indicate when data loss is imminent, as the following examples show.

Narrowing The Value Range

The program in Listing 23.13 is compiled by all development environments without any problems. The result appears to be 0, indicating a significant loss of information, as the result should be 0.75.

Listing 23.13: Implicit type conversion with data loss.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class NarrowingImplicit1Demo {
6
7    public static void main(String[] arguments) {
8      int a = 3;
9      int b = 4;
10     float result;
11     result = a / b;
12     System.out.println("Result = " + result);
13   }
14 }
```

The program performs an easily overlooked implicit conversion to an Int value when dividing. Since this data type can only store fixed-point numbers, the program simply truncates the division remainder. Only the digit before the decimal point is transferred. Thus, 0 appears as the result.

The second example (Listing 23.14) tests the limit of the value range. The program assigns two integer values their maximum value and then tries to add them together and store the result in the variable *result*.

Listing 23.14: A program runs amok.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class NarrowingImplicit2Demo {
6
7    public static void main(String[] arguments) {
8      int a = 2147483647;
9      int b = 2147483647;
10     int result;
11     result = a + b;
12     System.out.println("Result = " + result);
13   }
14 }
```

If you suspect the program will crash, you're wrong. The worst happens: the addition causes the intermediate result to be stored as an integer again, but this time it exceeds the maximum range of values. This does not cause the program to crash. Instead, you just get a completely wrong result; the program continues to work blithely.

Expanding The Range of Values

Implicit type conversion can result not only in a dangerous narrowing of the value range with a loss of data, but also in an extension of the value range (Listing 23.15). This time, the program stays on the rug.

Listing 23.15: Implicit type conversion with range expansion.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class ExpansionImplicitDemo {
6
7    public static void main(String[] arguments) {
8      byte a = 127;
9      byte b = 127;
10     int result;
11     result = a + b;
12     System.out.println("Result = " + result);
13   }
14 }
```

The example occupies two byte variables with values just below the maximum range of this data type. However, the addition does not cause the program to collapse because of the implicit conversion. Instead, the program converts the two values to an Int value, which is correctly passed to the variable *result*.

23.6.2 Explicit Conversion

For explicit type conversion, you must actively intervene in the program and perform an explicit conversion using the cast operator. In this respect, explicit conversion is nowhere near as tricky as the implicit variety. Here again there are two subtypes: the narrowing and the widening of the value range.

Narrowing of The Value Range

The narrowing of the value range should only be carried out if no loss of information occurs. It is an agreement between programmer and compiler that says, "It's okay, it's dangerous, but I know what I'm doing."

The example in Listing 23.16 shows how to narrow the range of values using an explicit type conversion. The program uses the cast operator to ensure that the result of the addition is interpreted as *short*. The result is just below the maximum value range. Therefore, everything goes well.

Listing 23.16: Tightening the value range by explicit type conversion.

```
1  //Examples/Rules
2
```

```
3  package programmingcourse;
4
5  public class NarrowingExplicitDemo {
6
7    public static void main(String[] arguments) {
8      int a = 30000;
9      int b = 2700;
10     short result;
11     result = (short) (a + b);
12     System.out.println("Result = " + result);
13   }
14 }
```

Expanding The Range of Values

Value range expansion refers to type conversion towards data types that take up more memory (Figure 23.6). Care should be taken when converting fixed-point numbers (*byte, short, int, long*) to floating-point numbers.

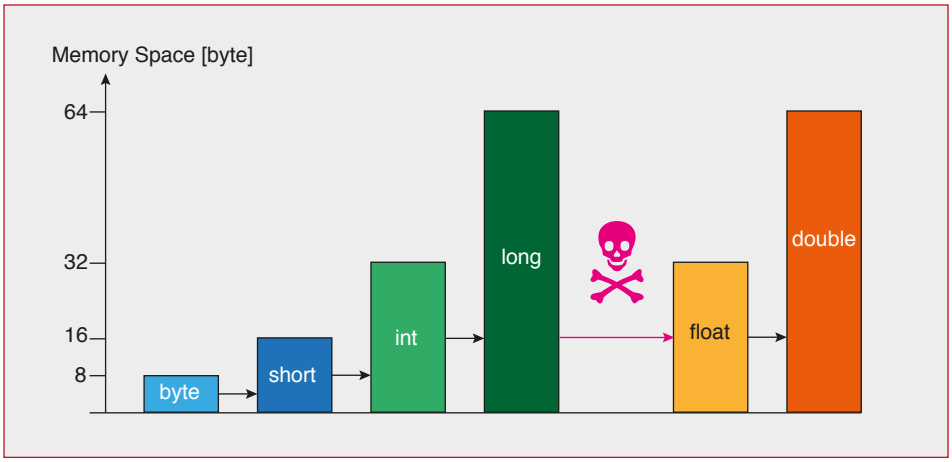


Figure 23.6: Caution should be taken when converting integers to commas.

The skull in the graphic indicates that the transition from fixed-point to floating-point numbers is dangerous. This is because, in principle, fixed-point numbers can always be stored only up to a certain decimal place. So, during the conversion, data loss may occur. Two examples for this: The first example shows how an extension of the value range can save the flow of the program (Listing 23.17).

Listing 23.17: Expanding the value range by explicit type conversion.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class ExpansionExplicit1Demo {
```

```
6
7     public static void main(String[] arguments) {
8       int a = 2147483647;
9       int b = 2147483647;
10      long result;
11      result = (long) a + (long) b;
12      System.out.println("Result = " + result);
13    }
14  }
```

While the previous example in Listing 23.14 collapses due to an implicit type conversion, this example pulls the emergency brake beforehand. The program performs an explicit conversion of the two summands and achieves the desired effect: the result is correct this time.

The following example shows that this does not always have to work so well. It demonstrates the already mentioned unpleasant side of the Java type conversion. This time, the extension of the value range leads to a loss of data (Listing 23.18).

Listing 23.18: Data loss due to value range extension.

```
1  //Examples/Rules
2
3  package programmingcourse;
4
5  public class ExpansionExplicit2Demo {
6
7    public static void main(String[] arguments) {
8      long a = 1234567890123456789L;
9      float b = a;
10     double c = a;
11     System.out.println("Long (64 Bit) = " + a);
12     System.out.println("Float (32 Bit) = " + b);
13     System.out.println("Double (64 Bit) = " + c);
14   }
15 }
```

The program outputs the following values:

```
Fixed point number (64 bit) = 1234567890123456789
floating point number (32 bit) = 1.23456794E18
Floating point number (64 bit) = 1.2345678901234567E18
```

The first line of the output shows the original value, while the second line shows the value converted to a floating point number of type *float*. The last line shows what happens when you convert the original value to a floating point number of type *double*. As you can see from outputs, the loss of precision is more or less significant depending on the memory area of the floating point numbers.

23.7 Polymorphism

Polymorphism means that an object can react differently depending on the situation. In Java, polymorphism can be realized by overloading and overwriting methods of a class.

23.7.1 Method Overloading

To demonstrate the overloading of methods, I would like to go back to the class Father. This time it should have two methods to specify the name of the object. The methods should both have the same name, but have different passing parameters. In one case the parameter is a char field, in the other a string (Listing 23.19).

Listing 23.19: Overloading a method.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    private String name;
8
9    public void setName(char[] privateArea) {
10     this.name = new String(privateArea);
11   }
12
13   public void setName(String privateArea) {
14     this.name = privateArea;
15   }
16
17   public String getName() {
18     return name;
19   }
20 }
```

The Java runtime environment distinguishes which of the two methods is used during program execution. If the char field is passed, the program executes the corresponding method. It appears, »My name is Karsten.« If, on the other hand, the string is passed, the program executes the method with the string parameter. It then displays: »My name is Prof. Karsten Roth.«

Listing 23.20: Two outputs by overloading methods.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Overloading1Demo {
6
```

```
7    public static void main(String[] args) {
8      char namePrivate[] = {'K','a', 'r', 's', 't','e', 'n'};
9      String nameProfessional = "Prof. Karsten Roth";
10     // Create objects:
11     Father karsten = new Father();
12     karsten.setName(namePrivate);
13     System.out.println("My name is " + karsten.getName() + ".");
14
15     karsten.setName(nameProfessional);
16     System.out.println("My name is " + karsten.getName() + ".");
17   }
18 }
```

So when a method of the same name is defined within a class, it is said to be overloaded. Any methods of the same type can be overloaded, including constructors. However, it is not allowed to overload methods of different types if the parameter list completely matches. Eclipse evaluates the following attempt as a duplicate definition and reacts with the error message *Duplicate method getName() in type father* (Listing 23.21).

Listing 23.21: Duplicate method getName() is not allowed.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    private String nameProfessional;
8    private char namePrivate[];
9
10   public void setName(char[] privateBereich) {
11     this.nameProfessional = new String(privateBereich);
12   }
13
14   public void setName(String privateBereich) {
15     this.nameProfessional = privateBereich;
16   }
17
18   public char[] getName() { // Compiler error
19     return namePrivate;
20   }
21
22   public String getName() { // Compiler error
23     return nameProfessional;
24   }
25 }
```

Overloading methods of different types is not allowed, otherwise the Java VM would not be able to distinguish which method is meant at runtime.

23.7.2 Method Overriding

An important aspect of biological inheritance is mutations. Even in Java, hereditary information does not have to be inherited just like that. If you want to change the behavior of a method in a derived class, you can simply overwrite (override) it. Overwriting is done by redefining the method. In doing so, the signature must be exactly the same as the base method. Methods cannot be overridden only if the base method is defined as *private*, *final* or *static* (Figure 23.7).

Figure 23.7: Overwriting methods is also bound to certain rules.

Overriding Normal Methods

Again an example with father and son. The class *father* is supposed to have a public method called *signature()*. It outputs title, name, and employer (Listing 23.22).

Listing 23.22: The class »Father« defines a method »signature()«.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    protected String name;
8    protected String title;
```

```
9     protected String employer;
10
11    public String getEmployer() {
12      return employer;
13    }
14
15    public void setEmployer(String employer) {
16      this.employer = employer;
17    }
18
19    public String getTitle() {
20      return title;
21    }
22
23    public void setTitle(String title) {
24      this.title = title;
25    }
26
27    public String getName() {
28      return name;
29    }
30
31    public void setName(String name) {
32      this.name = name;
33
34    }
35
36    public String signature() {
37      return title + " " + name + ", " + employer;
38    }
39  }
```

The child class is intended to represent the signature in a slightly different format. Compared to the signature of the *father* class, this signature is much shorter and omits the title (Listing 23.23).

Listing 23.23: The class »Child« overrides the method »signature()«.

```
1  //Examples/Rules
2
3  package roth;
4
5  /**
6   * Template for child objects like Katharina
7   */
8  public class Child extends Father {
9
10   public String signature() {
11     return name + " +++ " + employer + " +++";
12   }
13 }
```

The following program creates the two objects, assigns names and employers, and outputs the object's signature (Listing 23.24).

Listing 23.24: The daughter class overrides the base method »signature()«.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Family {
6
7    public static void main(String[] args) {
8
9      // Create objects:
10     Father karsten = new Father();
11     karsten.setName("Karsten Roth");
12     karsten.setTitle("Prof.");
13     karsten.setEmployer("University Mainz");
14     Child max = new Child();
15     max.setName("Max Roth");
16     max.setEmployer("Frankfurter Allgemeine");
17     // Display access:
18     System.out.println("The different signatures:");
19     System.out.println("Karsten (Father): "
20       + karsten.signature());
21     System.out.println("Max (Son): "
22       + max.signature());
23   }
24 }
```

The program produces the following output:

```
The different signatures:
Karsten (Father): Prof. Karsten Roth, University of Mainz
Max (Son): Max Roth +++ Frankfurter Allgemeine +++
```

In the first case, the program calls the original method of the parent class, and in the second case, it calls the overridden method of the child class. As can be seen, the derived class achieves a completely different program output by overwriting the base method.

Prevent Overriding

If you want to reliably prevent overwriting of a method, you must mark the method as *final*. Using the *signature()* method as an example, this works as seen in Listing 23.25.

Listing 23.25: prevent overwriting by using »final«.

```
1  //Examples/Rules
2
3  package roth;
4
5  public class Father {
6
7    protected String name;
8    protected String title;
9    protected String employer;
10
11   public String getEmployer() {
12     return employer;
13   }
14
15   public void setEmployer(String employer) {
16     this.employer = employer;
17   }
18
19   public String getTitle() {
20     return title;
21   }
22
23   public void setTitle(String title) {
24     this.title = title;
25   }
26
27   public String getName() {
28     return name;
29   }
30
31   public void setName(String name) {
32     this.name = name;
33
34   }
35
36   public final String signatur() {
37     return title + " " + name + ", " + employer;
38   }
39 }
```

The *final* keyword protects methods from being overwritten.

23.8 Summary

The visibility of classes can be specified in two levels, that of methods and variables in four levels. In addition to these visibility levels, the scope of variables is affected by blocks. Here an occlusion can occur, which overlaps class and object variables.

Java, as a typed language, requires an explicit conversion of the value range if data types do not match. This explicit conversion can result in data loss, so care must be taken here.

More insidious than the explicit conversion is the automatic (implicit) conversion of the type, where information can also be lost. You protect yourself against automatic conversion by converting intermediate results explicitly.

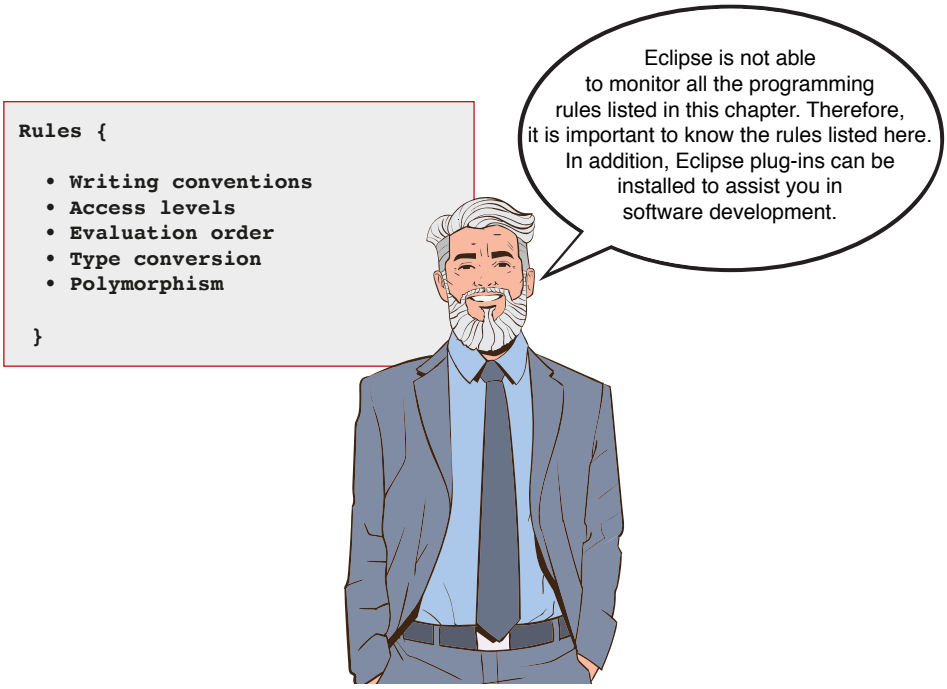


Figure 23.8: Even Prof. Roth doesn't know all the rules and regularities.

Overloading and overriding methods allows you to modify predefined classes via the means of inheritance. Overloading means defining methods with the same name with different parameters. When overwriting methods, you redefine them. They must have exactly the same signature for this.

23.9 Literature

Bernhard Steppan: Eclipse Extensions; *https://www.programmingcourse.net*

23.10 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/rules
There you will also find the solutions to the exercises.

24 Algorithms

24.1 Introduction

In the previous chapter, you saw the parts that constitute a Java program. If you put certain parts of a program together in such a way that a procedure for solving a problem is created, this is called an algorithm. How to develop such a procedure, which types of algorithms there are and which ones are already available in Java, that's what this chapter is about.

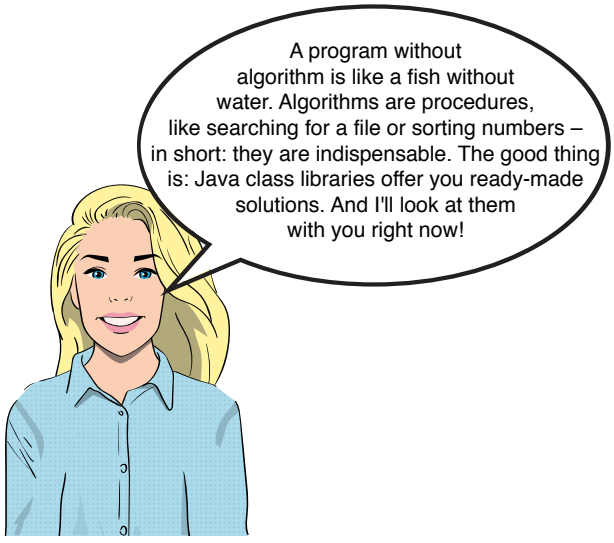


Figure 24.1: Anna will guide you through the world of algorithms.

24.2 Overview

24.2.1 Developing Algorithms

How to develop algorithms? To move purposefully from the problem to a solution, the algorithm, one should proceed according to a solution procedure. Such a plan could for example look like in figure 24.2.

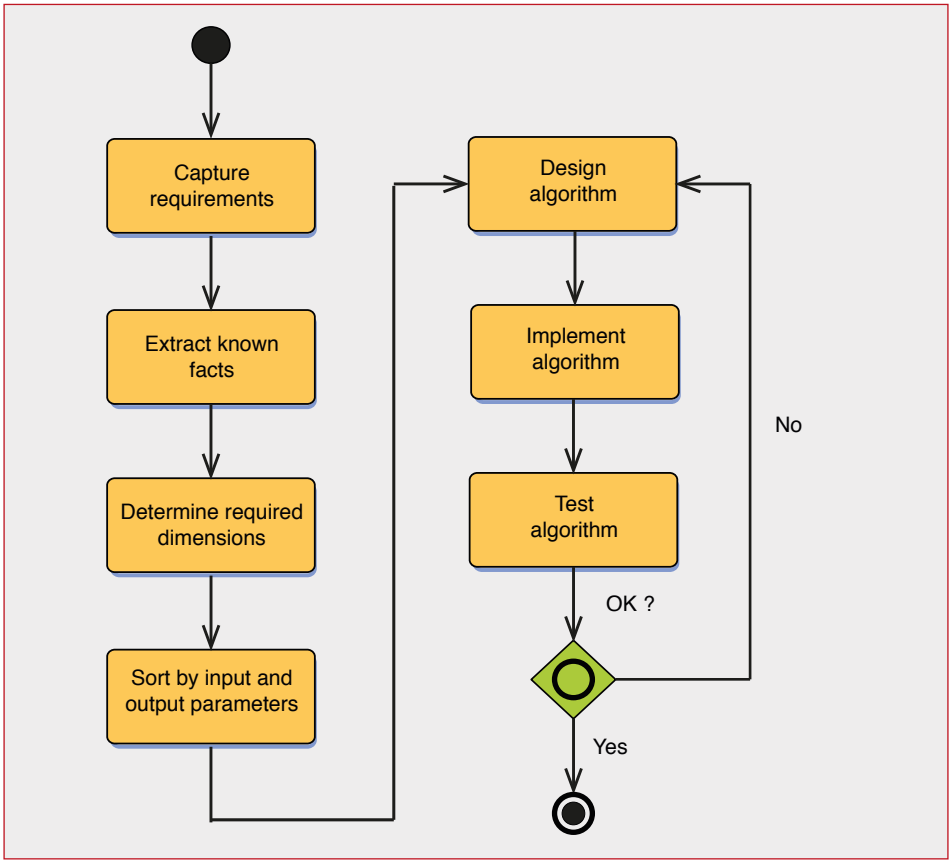


Figure 24.2: Sequence of the development of an algorithm.

Step 1 is usually called »Requirements Capture«. Steps 2 to 5 are called »Analysis and Design«, step 6 you know as »Implementation«, step 7 as »Test«. As you can see, the whole process is constantly repeated with each new requirement. The steps are cycled through in the following order: Requirements Capture – Analysis and Design – Implementation – Test.

The picture will surely look familiar to you from chapter 20, »Development Processes«. It is nothing else than the development process presented there. So what I have just presented to you here is a process for developing algorithms. In principle, the entire development of a program runs like this: What applies to the algorithm on a small scale also applies to the program on a large scale.

24.2.2 Types of Algorithms

Algorithms exist for every purpose. To better distinguish the abundance of algorithms, they are classified according to different types (Figure 24.3).

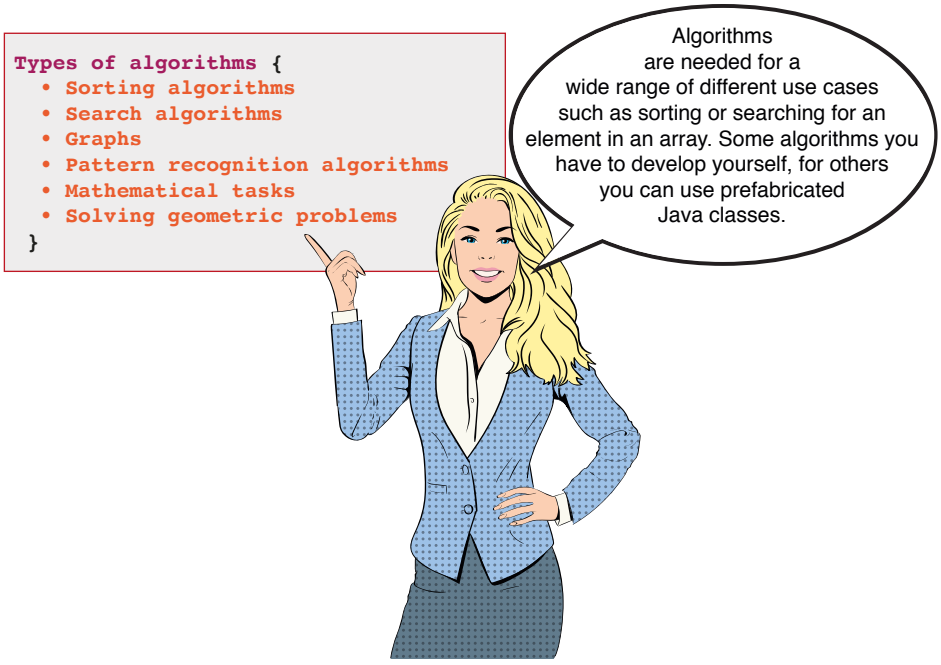


Figure 24.3: Types of algorithms.

24.2.3 Use of Algorithms

Algorithms can either be implemented by yourself or you can use prefabricated classes that have a suitable algorithm. For both cases I would like to present some examples below.

24.3 Develop Algorithms

24.3.1 Sorting Algorithms

It happens all the time in a program that values have to be sorted. These can be numerical values or character strings. The decisive factor for sorting algorithms is their speed. One of the simplest sorting algorithms is the *Selection Sort*.

Listing 24.1: One of the simplest sorting methods is the »Selection Sort«.

```
// Examples/Algorithms

package programmingcourse;

public class SelectionSortDemo {

  public static void main(String[] arguments) {

    int heights[] = { 185, 190, 171, 186, 184, 172 };

    int cache, index;

    for (index = 0; index <= (heights.length); index++) {
      for (int j = index; j < (heights.length); j++) {
        if (heights[index] > heights[j]) { // Comparison
          // Cache the minimum:
          cache = heights[index];
          heights[index] = heights[j]; // Minimum is overwritten
          // Swap by temporary value:
          heights[j] = cache;
        }
      }
    }

    System.out.println("The heights of the members of the "
        + "programming course sorted in ascending order:");
    for (index = 0 ; index < heights.length - 1; index++)
      System.out.print(heights[index] + ", ");
    System.out.print(heights[index]);
  }
}
```

The program first searches for the smallest element of the field, exchanges it for the first element, finds the second smallest element, exchanges it for the second element and so on until the entire field is sorted. In the innermost loop, the comparison between the elements takes place via an if statement. If a minimum is found, the program stores it in a temporary variable and swaps it. After that, more passes take place until the entire field is sorted. The program outputs the following:

```
The heights of the members of the programming course sorted in ascending
    order:
171, 172, 184, 185, 186, 190
```

24.3.2 Graphics Algorithms

There are different types of diagrams that are used, for example, in statistics programs or spreadsheets. One of these chart types is called bar chart and is very suitable to evaluate absolute values. For this chart type I would like to develop a simple algorithm with you here. The bar chart is to represent the sizes of the members of our programming course.

For drawing graphs using the class libraries of Java some previous knowledge is necessary. First, it is important to know that the Abstract Windowing Toolkit (AWT) provides a set of basic simple figures for drawing. Among them are circles, ellipses and rectangles. The figures can be filled or outlined. They are located in the class *Graphics* of the package *java.awt*.

24.3.2.1 Customized Graphics Component

To use these figures, you do not need to create an object of class *Graphics*. Instead, you derive a window component of type *JComponent* that provides the method *paint()* for drawing in a window. This method gets as parameter an object of class *Graphics*. This object has the method *fillRect()*, which can be used to draw rectangles (bars). To be able to draw your own diagrams, simply override the method *paint()* of the base class *JComponent*. By overriding the method, you replaced the behavior of the base class with your own implementation – the new graph algorithm.

The second important prerequisite for successfully implementing a graphics algorithm is knowledge of the Java coordinate system. The origin of the coordinate system is in the upper left corner. This means that the familiar coordinate system from mathematics, where you draw in the first quadrant, is mirrored on the x-axis (Figure 24.4).

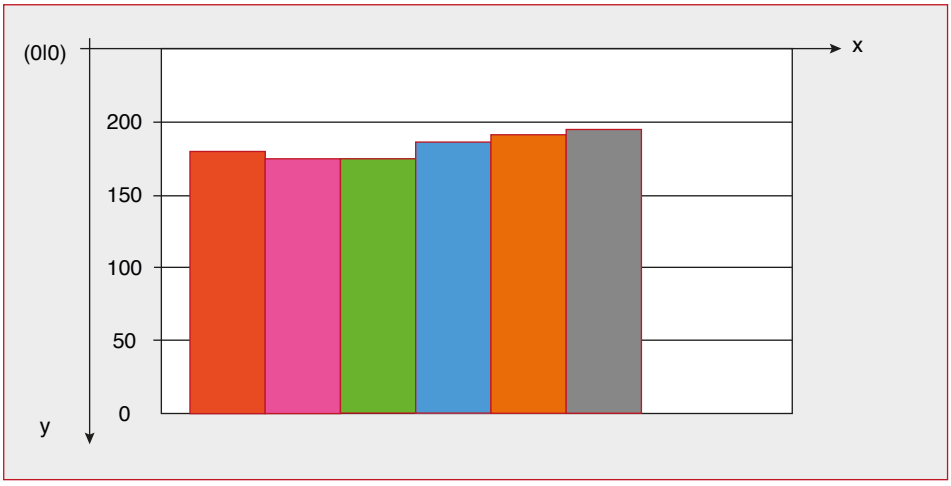


Figure 24.4: The coordinate origin is in the upper left corner.

24.3.2.2 Customized Chart Algorithm

Figure 24.5 shows even more precisely where the problem is when drawing a bar chart: Due to the tilted coordinate system, the *fillRect()* method always draws from top to bottom. A viewer of the chart expects exactly the opposite behavior. To make it look like all the bars grow upwards at a common point, you need to calculate the chart position on the y-axis correctly. It must be greater than the maximum value that a bar should display. The figure shows the chart position for values no greater than 250.

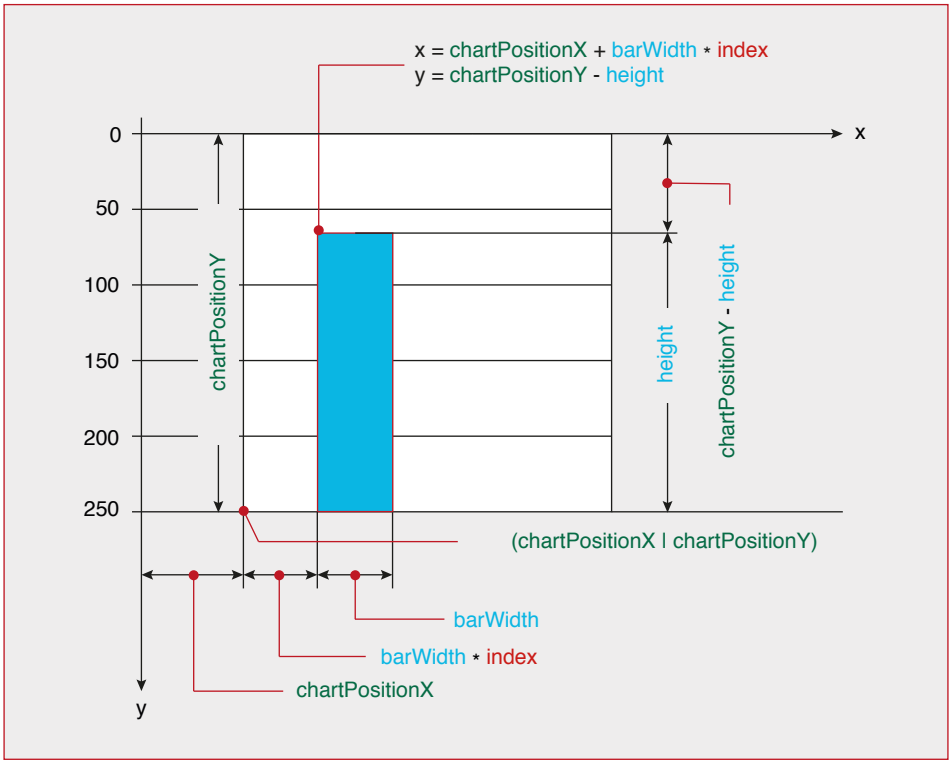


Figure 24.5: The calculation of the x and y coordinates of the bar chart.

The method *fillRect()*, which the program uses to draw the bars, takes four parameters to draw. Table 24.1 lists these parameters and shows you how the program calculates them.

Table 24.1: The parameters for the »fillRect()« drawing method.

Parameter	Purpose	Formula / Variable
x	Drawing origin in x-direction	chartPositionX + barWidth * index
y	Drawing origin in y-direction	chartPositionY - size
Width	Bar width	barWidth
Height	Size of a member	size

In the x-direction, the index and the bar width can be used to place the bars next to each other in an orderly manner. With index = 0 the drawing of the first bar starts at the value for *chartPositionX*, with *index = 1* shifted by the bar width to the right and so on. The best way to draw the bars side by side is to multiply the index by the bar width. The x parameter can be calculated according to the following formula:

```
x = chartPositionX + barWidth * index
```

Since the bars must be drawn in the negative direction in the y-direction, the y-coordinate of the drawing origin is calculated using the following formula:

```
y = chartPositionY - size
```

Two problems are solved by these formulas. In x-direction the algorithm places the first bar at the beginning of the diagram, because the index starts with the value 0. All further bars the algorithm offsets by the width of one bar in x-direction. In y-direction the algorithm places the bars so that they are on one line.

24.3.2.3 Implementation

First, we again need a class *person* to store the members of the programming course. The bar chart uses this class to generate a list of objects from it that corresponds to the programming course. The class is structured as in the other examples in the previous part of the book and contains the attributes *name* and *size*, which the bar chart will display later. According to the two attributes, there is a constructor that initializes the attributes and two query methods to get the values of the person (Listing 24.2).

Listing 24.2: This class provides the blueprint for the members of the programming course.

```
1  // Examples/Algorithms
2
3  package programmingcourse;
4
5  public class Person {
6    String name;
7    int height;
8
9    /**
10    * Constructor
11    * @param name the name of the person
12    * @param size the height of the person
13    */
14   public Person(String name, int height) {
15     super();
16     this.name = name;
17     this.height = height;
18   }
19
20   public String getName() {
21     return name;
22   }
23
24   public int getGroesse() {
25     return height;
26   }
27 }
```

Listing 24.3 shows the implementation of the graphics component. The class *bar chart* extends the Swing class *JComponent*, overriding the *paint()* method. The constructor starting at line 26 sets the default values for *diagramPositionX*, *diagramPositionY*, and the *barwidth*. Then the constructor assembles a field of objects of type *person* for the members of the programming course.

Starting at line 43, the overridden *paint()* method begins drawing the diagram. To begin, the method sets the font using the *setFont()* method of the *graphics* object of the *Graphics* class and the title color (lines 45 and 46). After that, the method outputs the title of the graph in line 49. The implementation of the graphics algorithm starts from line 52 with a For loop. The loop uses as a boundary value the length of the person array just created. Within the loop, the program varies the color of the bars so that they contrast in color (line 53). For this purpose, the program uses an array with predefined color values (lines 19 to 22).

In line 55, the program determines the size of the persons and draws the bars in the following lines 56 and 57. For drawing the bars the program uses the already mentioned method *fillRect()* of the class *Graphics*. The method gets the already mentioned parameters x, y, width and height, which are calculated according to the formulas from the figure 24.5. The labeling of the bars is done by the method *drawString()* in lines 59 to 61.

Listing 24.3: The class »BarChart« is the new graphics component.

```
1  // Examples/Algorithms
2
3  package programmingcourse;
4
5  import java.awt.Color;
6  import java.awt.Font;
7  import java.awt.Graphics;
8  import javax.swing.JComponent;
9
10 public class BarChart extends JComponent {
11   private static final long serialVersionUID = 1L;
12   private int chartPositionX;
13   private int chartPositionY;
14   private int barWidth;
15   private Color titleColor;
16   Person programmingCourse[] = new Person[6]; // Programming course
17
18   private Color[] colors = {
19     Color.red, Color.magenta, Color.green,
20     Color.blue, Color.orange, Color.gray
21   };
22
23   /**
24    * The component constructor
25    */
26   public BarChart() {
27     chartPositionX = 0;
28     chartPositionY = 0;
29     barWidth = 80;
30     programmingCourse[0] = new Person("Karsten", 184); // 184 cm
31     programmingCourse[1] = new Person("Anna", 171); // 171 cm
32     programmingCourse[2] = new Person("Julia", 172); // 172 cm
33     programmingCourse[3] = new Person("Lukas", 186); // 186 cm
34     programmingCourse[4] = new Person("Florian", 185); // 185 cm
```

```
35      programmingCourse[5] = new Person("Robert", 190); // 190 cm
36    }
37
38    /**
39     * The paint() method is overridden by the base class.
40     * This is the method that contains the graph algorithm
41     * that draws the diagram.
42     */
43    public synchronized void paint(Graphics grafik) {
44      //Title fint:
45      grafik.setFont(new Font("Arial", Font.PLAIN, 12));
46      grafik.setColor(titleColor); // Title color
47      // Chart title
48      grafik.drawString("Size comparison of the members "
49          + "of the programming course", 145, 20);
50      int height;
51      // The algorithm for drawing rectangles:
52      for (int index = 0; index < programmingCourse.length; index++) {
53        grafik.setColor(colors[index]);
54        height = programmingCourse[index].getGroesse();
55        // Drawing the rectangles:
56        grafik.fillRect(chartPositionX + barWidth * index,
57          chartPositionY - height, barWidth, height);
58        // Labeling with the name of the members:
59        grafik.drawString(programmingCourse[index].getName(),
60          chartPositionX + 20 + barWidth * index,
61          chartPositionY + 20);
62      } // for
63    } // paint
64
65    public void setTitleColor(Color titleColor) {
66      this.titleColor = titleColor;
67    }
68
69    public void setBarWidth(int barWidth) {
70      this.barWidth = barWidth;
71    }
72
73    public void setChartPositionX(int chartPositionX) {
74      this.chartPositionX = chartPositionX;
75    }
76
77    public void setChartPositionY(int chartPositionY) {
78      this.chartPositionY = chartPositionY;
79    }
80  }
```

Now, it would still be desirable for the graph class to get a small interface to be able to change some parameters from the outside. It comes in the form of three access methods that allow you to vary the character origin, the height of the chart, and the width of the bar:

```
public void setBarWidth(int barWidth)
public void setChartPositionX(int diagramPositionX)
```

```
public void setChartPositionY(int diagramPositionY)
```

The *chartWindow* class embeds the chart component in a Swing class (Listing 24.4). To do this, it creates a new object named *barChart* at line 16. In the window's initialization method (starting at line 30), the program gets a drawing area and sets a border layout. Following this, the window sets its size and window title. With the *setChartPositionX()* method, the window moves the origin of the bar chart on the x-axis slightly to the right, and with the *setChartPositionY()* method, it specifies the maximum total height of the chart and thus determines the line on which all bars are placed. Finally, the class inserts the chart into the *drawingArea* with the *add()* method (line 38).

Listing 24.4: The Graphics Window class embeds the bar chart in a window.

```
1  // Examples/Algorithms
2
3  package programmingcourse;
4
5  import java.awt.AWTEvent;
6  import java.awt.BorderLayout;
7  import java.awt.Dimension;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10
11
12 public class GraphicsWindow extends JFrame {
13   private static final long serialVersionUID = 1L;
14   private JPanel drawingArea;
15   private BorderLayout layout = new BorderLayout();
16   private BarChart barChart = new BarChart();
17
18   // Create the graphics window
19   public GraphicsWindow() {
20     enableEvents(AWTEvent.WINDOW_EVENT_MASK);
21     try {
22       init();
23     }
24     catch(Exception e) {
25       e.printStackTrace();
26     }
27   }
28
29   // Components initialization
30   private void init() throws Exception  {
31     drawingArea = (JPanel) getContentPane();
32     drawingArea.setLayout(layout);
33     setSize(new Dimension(600, 400));
34     setTitle("GraphicsApp");
35     barChart.setAlignmentX((float) 0.5);
36     barChart.setChartPositionX(55);
37     barChart.setChartPositionY(300);
38     drawingArea.add(barChart,  BorderLayout.CENTER);
```

```
39      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40    }
41  }
```

The main program can now generate the completed main window. This is done right at the beginning of its constructor on line 14. The object named *window* has the various methods that can be used to create windows. One method is *setResizeable()*, which the program calls at line 21 and which causes the user not to be able to resize the window. Then the window is centered on the screen with the *setLocationRelativeTo(null)* method (line 22) and made visible with the *setVisible(true)* method (line 23).

Listing 24.5: The class »GraphApp« is the graphics program.

```
1  // Examples/Algorithms
2
3  package programmingcourse;
4
5  import javax.swing.UIManager;
6
7  public class GraphicsApp {
8    boolean packFrame = false;
9
10   /**
11    * Constructor
12    */
13   public GraphicsApp() {
14     GraphicsWindow window = new GraphicsWindow();
15     if (packFrame) {
16       window.pack();
17     }
18     else {
19       window.validate();
20     }
21     window.setResizable(false);
22     window.setLocationRelativeTo(null); // Center window
23     window.setVisible(true);
24   }
25
26   /**
27    * The method main is the program's entry point.
28    * @param args the arguments are not evaluated
29    */
30   public static void main(String[] args) {
31     try {
32      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
33     }
34     catch(Exception e) {
35       e.printStackTrace();
36     }
37     new GraphicsApp();
38   }
39 }
```

When you compile and run the program, you will see the main window containing the new bar chart (Figure 24.6). In the upper part of the graph you can see the title, in the middle the bar graph and in the lower part the label of each bar with the name of the person.

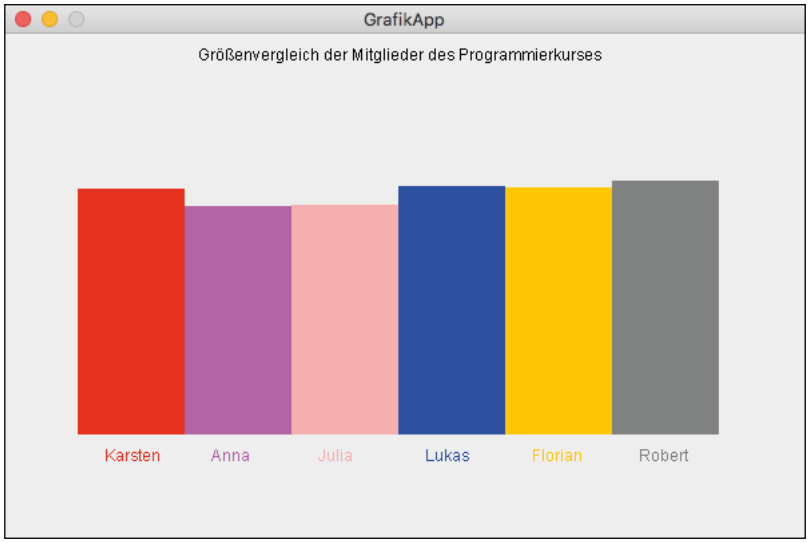


Figure 24.6: The size of the members of the programming course as a bar chart

24.4 Algorithm Usage

As a Java programmer, you do not have to develop many algorithms yourself. You have seen in chapter 22, »Class Libraries«, that the class libraries of the Java Standard Edition can facilitate your programming. Inside these class libraries are classes that contain, for example, sorting algorithms. You just have to know if it is suitable for your purpose. But also for this an example.

24.4.1 Sorting Algorithms

24.4.1.1 Collections

In chapter 22, Class Libraries, you were introduced to a number of class libraries, including the collection classes that are in the util package of the Java Standard Edition. There are quite a few different methods. On the subject of sorting, I would like to introduce you to the class *Collections*. It contains a method that is excellent for sorting fields, as the following example shows.

Listing 24.6: Sorting with the help of the Collections class.

```
// Examples/Algorithms

package programmingcourse;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsSortDemo {

  public static void main(String[] arguments) {

    List<String> programmingCourse =
      Arrays.asList(new String[] {"Karsten", "Anna", "Julia",
                                  "Lukas", "Florian", "Robert"});
    Collections.sort(programmingCourse);
    System.out.println("The first names of the members of the "
        + "programming course sorted alphabetically:");
    int index;
    for (index = 0 ; index < programmingCourse.size() - 1; index++)
      System.out.print(programmingCourse.get(index) + ", ");
    System.out.print(programmingCourse.get(index));
  }
}
```

The program creates an object called *programmingCourse*, which is passed a string array containing the names of the members of the programming course. The class sorts this array using the *Collections* class and presents the alphabetically sorted list. Unlike the example of the listing 22.17 from chapter 22, »Class Libraries«, this time the program outputs the names of the members of the programming course in order:

```
The first names of the members of the programming course sorted
    alphabetically:
Anna, Florian, Julia, Karsten, Lukas, Robert
```

24.4.1.2 Arrays

Just how much the sorting program of the listing 24.1 can be simplified if you know the power-ful Java class libraries is shown by the listing 24.7.

Listing 24.7: Sorting using the class »Arrays«.

```
// Examples/Algorithms

package programmingcourse;

import java.util.Arrays;

public class ArraysSortDemo {

  public static void main(String[] arguments) {
```

```
10
11      int heights[] = { 185, 190, 171, 186, 184, 172 };
12      int index;
13
14      Arrays.sort(heights); // Sorting the field
15      System.out.println("The heights of the members of the"
16          + "programming course sorted in ascending order:");
17      for (index = 0 ; index < heights.length - 1; index++)
18        System.out.print(heights[index] + ", ");
19      System.out.print(heights[index]);
20    }
21  }
```

As you can see, a simple method call to the *Arrays* class replaces the entire loop nest of the 24.1 listing. It outputs exactly the same sorted list of the sizes of the members of the programming class:

```
The heights of the members of the programming course sorted in ascending
    order:
171, 172, 184, 185, 186, 190
```

The example shows well that you only have to come up with algorithms yourself for which Java does not provide any ready-made solutions. This is also true for searching, for example, as the following example will show.

24.4.2 Search Algorithms

If you manage large lists of objects, you naturally want to find individual objects again without problems. The following listing 24.8 is an extension of the previous example. This program first creates our programming course with the individual members again (lines 12 and 13). After that, the program sorts the list, again using the *Collections* class (Line 16).

Listing 24.8: The search for »Florian« within the programming course.

```
1  // Examples/Algorithms
2
3  package programmingcourse;
4
5  import java.util.Arrays;
6  import java.util.Collections;
7  import java.util.List;
8
9  public class CollectionsSearchDemo {
10
11   public static void main(String[] arguments) {
12     // Creating the programming course with its members
13     List<String> programmingCourse =
14       Arrays.asList(new String[] {"Karsten", "Anna", "Julia",
15                                   "Lukas", "Florian", "Robert"});
16     // Sorting the programming course
```

```
17      Collections.sort(programmingCourse);
18
19      int index; // Running index for loop and search
20
21      // Output of the sorted list of members:
22      System.out.println("The first names of the members of the "
23          + "programming course sorted alphabetically:");
24      for (index = 0 ; index < programmingCourse.size() - 1; index++)
25        System.out.print(programmingCourse.get(index) + ", ");
26      System.out.print(programmingCourse.get(index) +"\n");
27
28      // Search for member "Florian":
29      index = Collections.binarySearch(programmingCourse, "Florian");
30
31      // Determination of the element by means of index & output its name:
32      System.out.print("Member found: " + programmingCourse.get(index));
33    }
34  }
```

The *Collections* class has a static method *binarySearch()* that the program passes the list of members of the programming course and the member it should search for (line 27). In this example, the program is to determine if there is a person named »Florian« in the programming course and output the *index* of that person. To display the person, the program uses the *get()* method of the *programmingcourse* object. The program outputs the following:

```
The first names of the members of the programming course sorted
     alphabetically:
Anna, Florian, Julia, Karsten, Lukas, Robert
Member found: Florian
```

24.5 Summary

A procedure for solving a problem is called an algorithm. In this chapter, you have seen several examples of homegrown procedures. The first example showed a complicated sorting algorithm that can be implemented much more easily using Java class libraries.

The following graphics algorithm has shown what an own method for drawing a bar chart looks like. However, for most such problems there are ready-made solutions in the form of class libraries. Therefore, it is good to get an overview of possible mature solutions before programming complex procedures.

24.6 Literature

Robert Sedgwick, Kevin Wayne: Algorithms in Java, Pearson Studium, 4. Auflage, 2014, Hallbergmoos

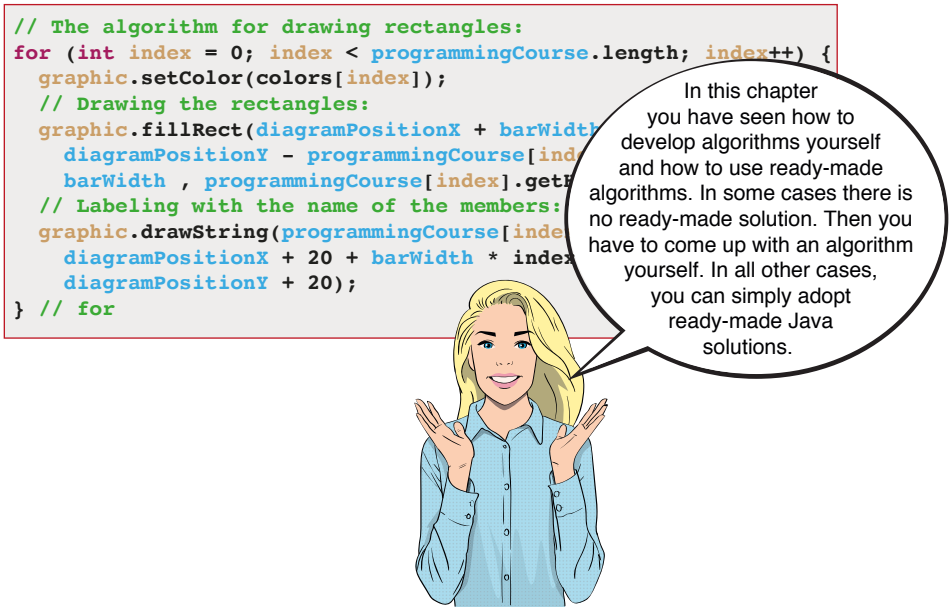


Figure 24.7: The Java class libraries provide many ready-made algorithms.

24.7 Exercises

When you have finished this chapter, please go the exercises to deepen your knowledge:
https://www.programmingcourse.net/books/java_with_eclipse/algorithms
There you will also find the solutions to the exercises.

PART IV

Java Projects

In my experience, Java beginners often have the problem that they understand very simple example programs, but do not know how to build larger programs. Most of the projects you encounter in companies are far too large and poorly documented for learning Java.

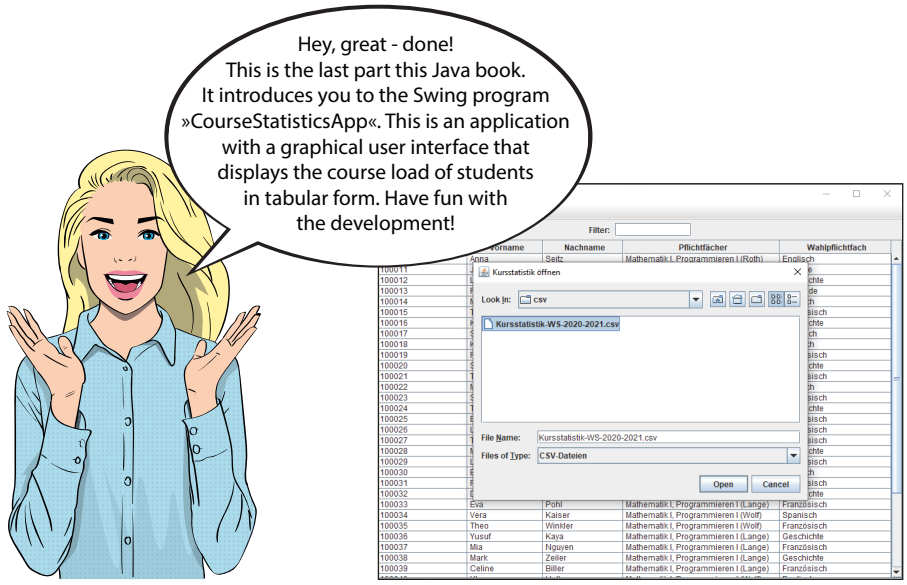


Figure 24.8: CourseStatisticsApp, a program to output the course enrollment of a semester.

So the ideal would be a manageable Java project that a Java beginner can understand – that's exactly what this book part presents to you on a single project. The book part consists of only one project called »CourseStatisticsApp«, which is developed from A to Z with Eclipse. You will once again get to know all the elements of a Java program in context using this manageable example.

25 Swing Programs

25.1 Introduction

The conclusion of this book consists of a bigger program with a graphical user interface. It is, again, intended to tabulate course statistics from our programming course. With four classes, the project is much larger than the previous examples, but still suitable for beginners. It shows how larger Java programs are constructed. The user interface consists of elements that come from the Swing class library, which you already know from chapter 22, »Class Libraries«. Swing has the advantage that it is part of your Java installation and you do not have to install anything extra.

Figure 25.1: Professor Roth and his programming class say goodbye.

25.2 Requirements

The program *CourseStatisticsApp* should tabulate a list of students in a semester. The program should have only one menu named FILE with the commands OPEN and EXIT. Below the file menu, there shall be an input field to filter the list. The field should allow to reduce the list by a search term. Below the field there shall be a table with five columns (Figure 25.2).

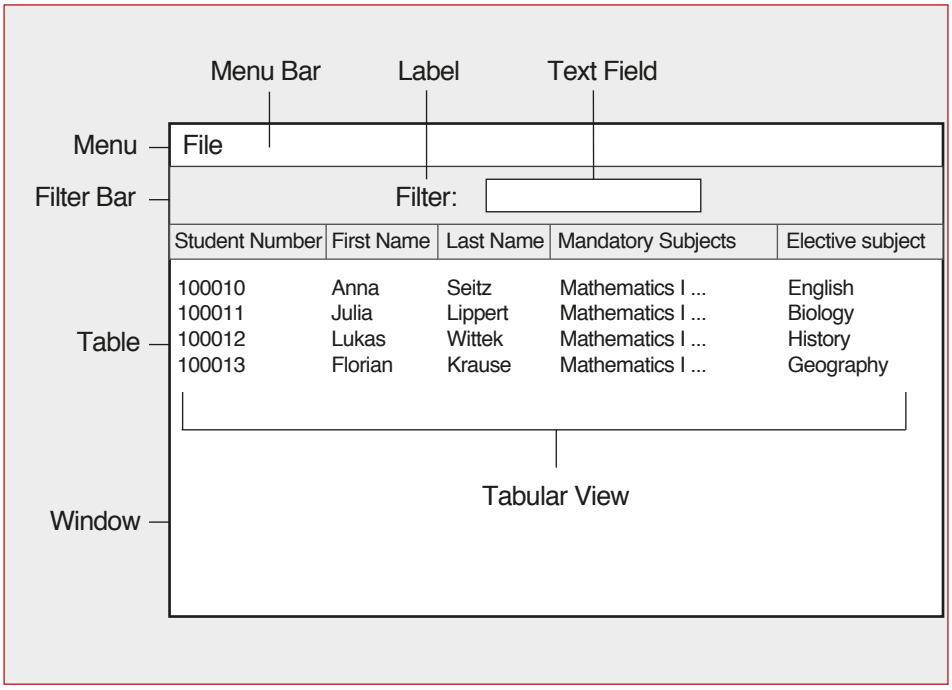


Figure 25.2 The requirements for the user interface and presentation of the CSV file

The first column contains the student's matriculation number, the second the first name, the third the last name, the fourth the majors, and the last column the elective. The list is stored in the form of a CSV file on a hard disk. In CSV files, the values are separated from each other with a separator. In the German language area one uses the semicolon for this. The program should be able to read the file and display the values in tabular form in the way mentioned before.

25.3 Analysis and Design

The program breaks down into two technically distinct sections: the graphical user interface and the program logic such as reading a CSV file and the filter function. Let's start with the user interface.

25.3.1 User Interface

The user interface of the program basically consists of the main window. The structure of this main window resembles a picture with a frame (Figure 25.3). The window itself is the frame with a background. A menu bar and two larger elements are integrated on it. The menu bar consists of a file menu, which contains two menu commands. With the first menu command OPEN... a file can be opened. The word OPEN is followed by a so-called ellipsis. These three dots signal that the command will open a dialog. The second command EXIT terminates the program.

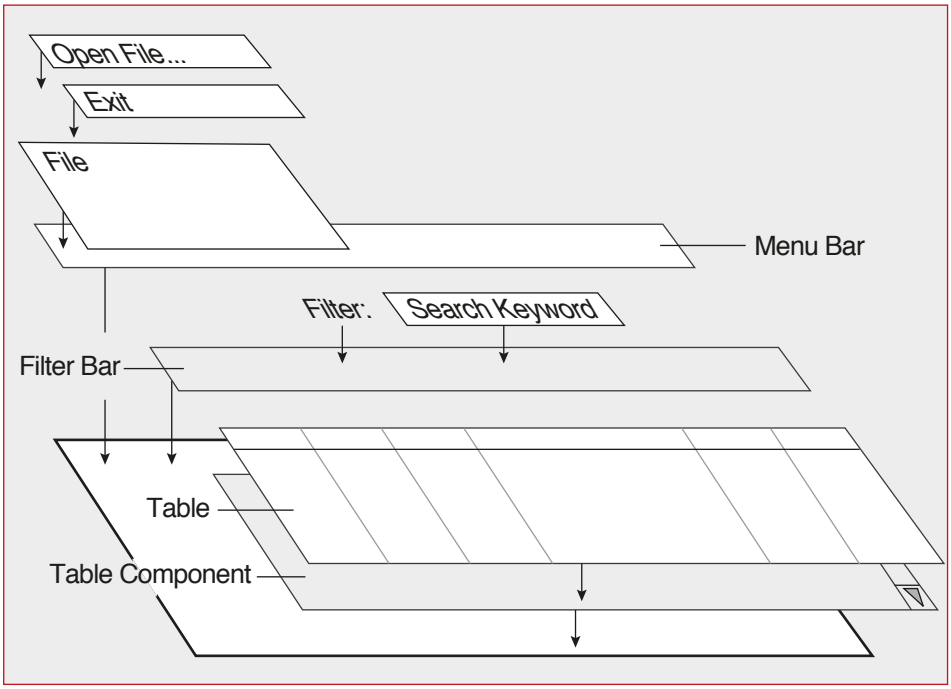


Figure 25.3: The structure of the main window.

The first element at the top of the window is a bar. It is used to integrate a text field for a filter and the corresponding label into the main window. In the lower area of the window there is an element that literally translates as a scroll disk. The table is placed on this element. It is a peculiarity of the Swing class library that tables must be added to a scrolling element. If this is omitted, their contents cannot be scrolled.

25.3.2 Program Logic

Overview

Figure 25.4 shows an overview of the entire program flow. When you start the program, it waits for input. Two types of input are possible: either you open a file or you exit the

program. Use the menu command OPEN to open a file. A dialog will then appear allowing you to search for a CSV file for the program to display.

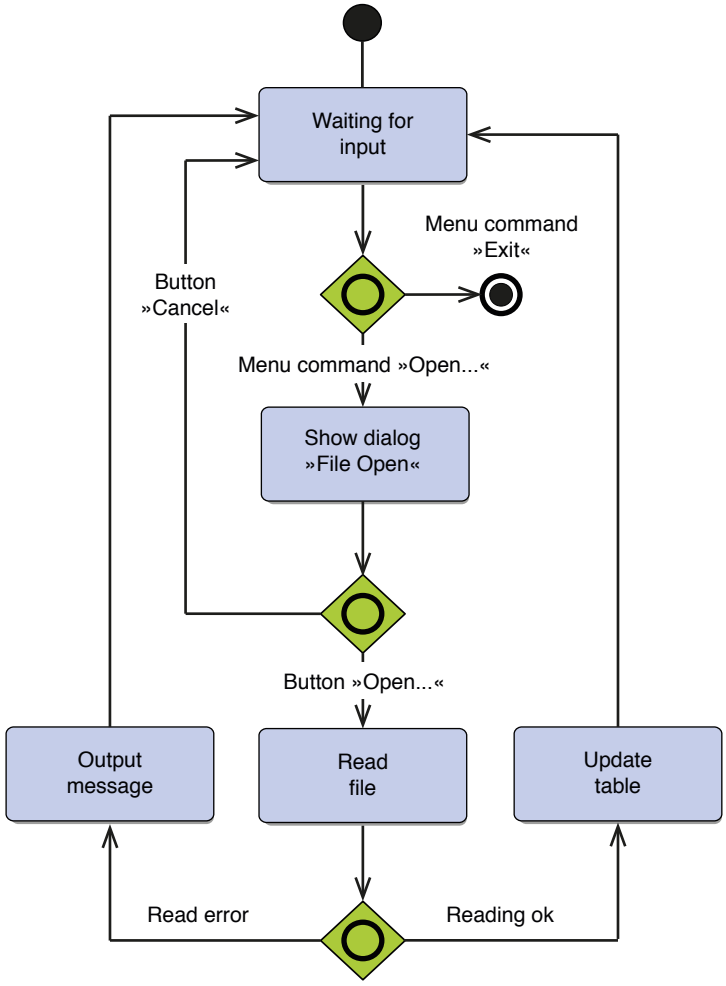


Figure 25.4: The process diagram of the overall flow.

Once you have found and selected a file, you can use the OPEN button to load that file. If this is successful, the program will update the table view. If not, the program gives an error message and then waits for a new input. The program will also wait for an input if you click the CANCEL button of the OPEN FILE dialog.

Reading the File

A CSV file is a very simple text file. It is organized like a table. The rows of this table are called records. Each record consists of individual data fields, the columns of a table (Figure 25.5). The records are separated by one or two break characters (this depends on the op-

erating system). The usual characters are *Carriage Return* and *Line Feed*, which still come from the typewriter era.

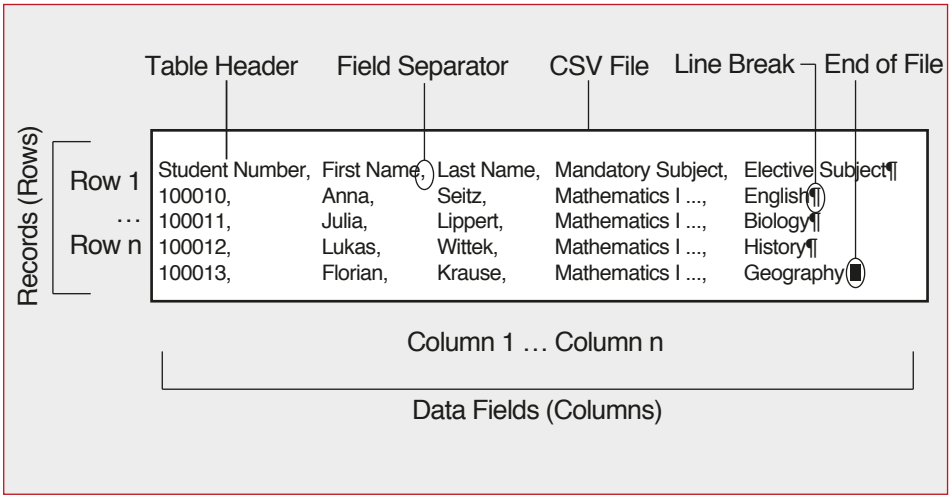


Figure 25.5: The structure of the CSV file.

In order for a program to distinguish the data fields of a record when reading the CSV file, they must be separated with a character that is known to the program before reading. In principle, many characters such as a comma, colon or semicolon can be used. It is only important that the character is not contained in the values of the data fields. In the English language area a comma has established itself as a separator. This separator also explains the original meaning of the abbreviation CSV, which stands for »Comma-separated values«.

CSV files are often used to exchange numerical values. Unfortunately, the representation of comma-separated values is not standardized internationally. For example, in the United States, decimal places are separated by a period, whereas in the German-speaking world, they are separated by a comma. Comma numbers according to German formatting could not be interpreted correctly with a comma as a separator, which is why we have established a semicolon as a separator for data fields. This should also apply to the structure of the quote list.

The reading of a CSV file requires a so-called parser (English: to parse = analyze, dissect). A parser breaks down a file into its individual components. To do this, the parser reads in each line of a file. With the help of the separator for the individual values, the parser splits each line into its components. By assigning the values, the rows and columns of a table are created. The parser repeats the process of reading until it reaches the end of the file.

Developing the Data Model

To map the CSV data structure to Java classes, it is useful to develop a data model before implementing the Java program. The CSV file consists of an unknown number of records, the rows of the later table. Each of these rows in turn consists of an unknown number of data fields, the columns of the later table (Figure 25.6).

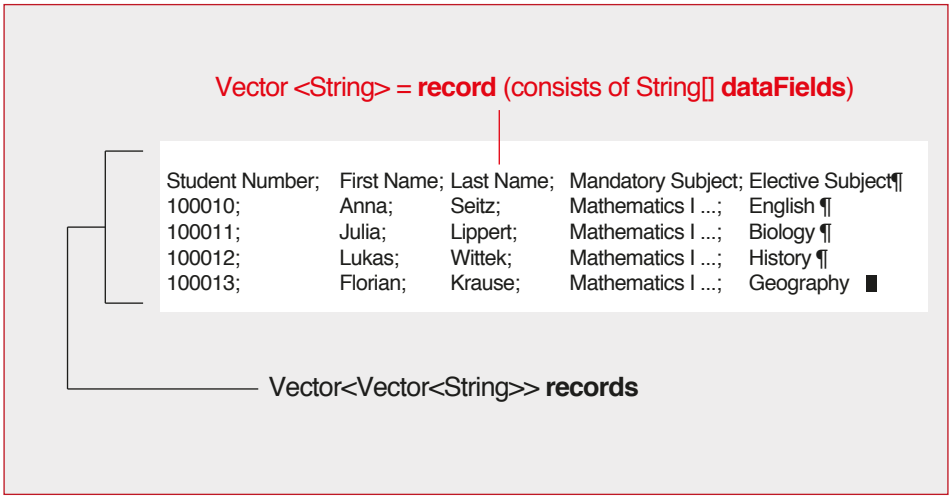


Figure 25.6: The course statistics data model.

If it is not clear how large a table can grow in the x or y direction, you must use a dynamic array for the table's data model. Dynamic arrays can be implemented, for example, with the *Vector* class. The class *Vector* also harmonizes very well with swing tables. Each data field of such a swing table is an element of the type *String*. Together they result in a record of type *Vector<String>* (one-dimensional array). Finally, all records result in a variable named *datasets*, which is of type *Vector<Vector<String»* – a two-dimensional array.

Filtering the Table

When the CSV file is read in and presented as a table, the user often wants to get information about a particular person or about a particular course in a semester. There are two options here: searching and filtering. With a search, the program would simply highlight a found text passage, but continue to display the rest of the table. A filter, on the other hand, hides all rows where there is no hit. For this reason, the program *CourseStatisticsApp* is to be equipped with a filter.

25.4 Implementation

25.4.1 Start Eclipse With the Workspace »Exercises«

Start Eclipse again as usual with the workspace named *Exercises*. If you have switched the workspace in the meantime, return to the *Exercises* directory and load Eclipse with that workspace.

25.4.2 Create New Java project »Swing Programs«

Then call the wizard to create a new Java project. In this dialog, enter the title of this chapter *Swing Programs* as the project name. Then check again that the dialog shows at least the Java 19 as Java Runtime Environment (JRE) and uncheck the option CREATE MODULE-INFO.JAVA FILE on the second page.

25.4.3 Create New class »CourseStatisticsApp«

Now expand the new project in the PACKAGE EXPLORER on the left side of Eclipse. Inside the project is the node *src*. Then right-click on the directory and select the menu command NEW → CLASS. Eclipse will start a dialog to create a new class. In the PACKAGE field, enter the name of the package *net.programmingcourse.coursestatistics.main*.

Fill the NAME text box with the name of the class *CourseStatisticsApp*. After that, select that Eclipse creates the *main* method. Finally, click FINISH to close the dialog and let Eclipse create the class. Of course, the class is still practically empty. You will need to implement it in the next step.

25.4.4 Implement Class »CourseStatisticsApp«

Implement the class as shown in Listing 25.1. To do this, a new object of type *MainWindow* must be created in line 15. You will later derive the class *MainWindow* from the Swing class *JFrame* and use it to inherit all the methods you will call in the next section. This window will later take on the role of the program's graphical user interface.

Listing 25.1: The main program of the course statistics.

```
1  //Examples/Swing_Programs
2
3  package net.programmingcourse.coursestatistics.main;
4
5  import java.awt.Dimension;
6  import javax.swing.JFrame;
7  import net.programmingcourse.coursestatistics.ui.MainWindow;
8
9  /**
10  * Project: CourseStatisticsApp
11  * Description: The main app for displaying course statistics
12  * @author Bernhard Steppan
13  * @version 1.0
14  *
15  */
16 public class CourseStatisticsApp {
17
18   public static void main(String[] args) {
19     MainWindow window = new MainWindow();
20     // Set the size of the main window:
```

```
21      window.setSize(new Dimension(800, 600));
22      // Centre window (= parameter null):
23      window.setLocationRelativeTo(null);
24      // Set that the app is terminated when the window is closed:
25      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26      // Show window:
27      window.setVisible(true);
28    }
29  }
```

Using the method *setLocationRelativeTo()* with the parameter *null* in line 19, the window is centered on the screen. The method is actually universally applicable to display the window relative to another component if you pass a concrete value to the method. However, if you pass the parameter *null* instead of a concrete value, the window will center itself on the screen.

Calling the *setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)* method causes the program to close by clicking the window's close cross. This behavior is not desirable in every case. However, in this case, where the program consists only of the main window, it makes a lot of sense.

Finally, on line 23, the program calls the method *setVisible()* with the parameter *true* to make the window appear on the screen. If you have implemented everything according to listing 25.1, now save the project.

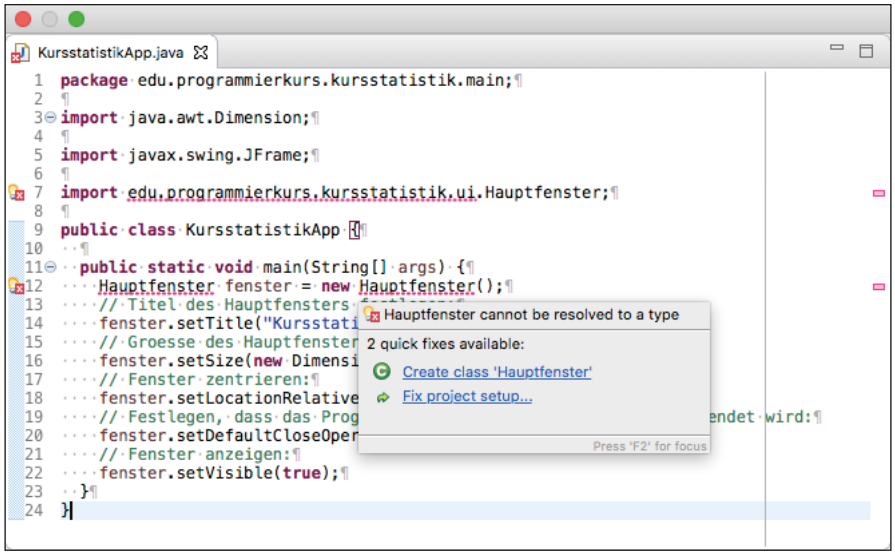


Figure 25.7: Create New Class »MainWindow«.

25.4.5 Create New Class »MainWindow«

Eclipse will then mark the *MainWindow* class as unknown in the editor. Place the cursor on the class name in the Eclipse editor and wait until the editor displays a pop-up window

(Figure 25.7). From the pop-up window, choose CREATE CLASS 'MAINWINDOW'. Eclipse will then start a dialog to create this class.

In the PACKAGE field of the dialog, enter the name *net.programmingcourse.coursestatistics.ui*. This is supposed to be the package where everything related to the graphical interface of the program will be located later. The class *mainwindow* must be derived from the Swing class *JFrame*. In the Swing class library, the *JFrame* class is the class used to create windows of this type. You can achieve the inheritance by inserting *Superclass* in the *javax.swing.JFrame* field of the dialog.

If you ever don't know the exact name of the class, you can search for it by clicking the *Browse...* button. When everything is complete, click FINISH to have Eclipse create the class. The Eclipse editor in the right part of the development environment should display the source code of the listing 25.2. This class is also almost empty and still needs to be implemented.

Listing 25.2: The class »MainWindow« created by Eclipse.

```
1 package net.programming.coursestatistics.ui;
2
3 import javax.swing.JFrame;
4
5 public class MainWindow extends JFrame {
6
7 }
```

25.4.6 Implementing Class »MainWindow«

Insert Constructor and Initialization Method

Now, in the Eclipse text editor, insert the main window constructor and an initialization method (Listing 25.3). The constructor consists of calling the constructor of the *JFrame* class using *super()* on line 12, followed by a logging command that I will discuss later. On line 14 comes a method that creates the window's graphical interface. This process is called initialization, hence the name *init()* of this method. Lines 17 and 18 show the body of this method, which the next sections of this chapter will fill with interface elements.

Listing 25.3: The constructor of the main window with the body of the method »init()«.

```
1  package net.programmingcourse.coursestatistics.ui;
2
3  import java.util.logging.Logger;
4  import javax.swing.JFrame;
5
6  public class MainWindow extends JFrame {
7
8    private static final long serialVersionUID = 1L;
9    private static final Logger logger =
10     Logger.getLogger("CourseStatistics");
11
```

```
12    public MainWindow() {
13      super();
14      logger.info("Main window is created");
15      init();
16    }
17
18    private void init() {
19    }
20  }
```

Logging is essential for professional programs for debugging. It creates a kind of log of the program. Java provides the *Logger* class for this purpose, which is imported in line 3. Add this import statement and the static variable on line 9, and the class should be error-free except for one warning.

Eclipse displays the warning because the class is missing an ID for serialization. Any Java class that can be serialized must contain such an ID. Serialization allows the class to be stored on a disk and reloaded. Add the static variable as seen in line 8. This should also make the class free of errors and warnings. This is followed by another constant called PROGRAM_NAME, which the program will need later for the title of the window and for logging output.

Now start the program again. An empty window appears again, but with the difference that in Eclipse's CONSOLE window the text *Main window is created* appears. The logging information appears as before commands of type *System.out.println* in the console window of Eclipse. For logging to be helpful in debugging, the programmer must insert logging commands into his source code at meaningful places. This chapter will show you exactly how this works.

Implementing method »init()«

Now complete the method *init()* according to listing 25.4. To do this, complete the logging statement on line 34, which generates an information that the program generates the graphical interface. After that, the program sets the title of the window using the *setTitle()* method, which takes the previously defined constant *PROGRAM_NAME* for the name of the program as a parameter. This method belongs to the base class *JFrame* and can therefore be called here.

The line below it sets the program's menu bar via the *setJMenuBar()* method. The method was inherited from the base class *JFrame* like the method before. What is unusual here is the parameter, which is a method call. The *createMenuBar()* method creates the complete menu bar and returns it as a parameter to the *setJMenuBar()* method. You complete this method following the initialization method.

Listing 25.4: The method »init()« creates the components of the program interface.

```
1  package net.programmingcourse.coursestatistics.ui;
2
3  import java.awt.BorderLayout;
4  import java.util.logging.Logger;
5  import javax.swing.JFrame;
```

```
6
7  /**
8   * Main window of the program "Course Statistics"
9   *
10  * @author Programming Course Prof. Roth
11  *
12  */
13 public class MainWindow extends JFrame {
14
15   private static final long serialVersionUID = 1L;
16   private static final String PROGRAM_NAME = "Course Statistics";
17   private static final Logger logger =
18       Logger.getLogger("CourseStatistics");
19
20   /**
21    * Constructor of the main window
22    */
23   public MainWindow() {
24     super();
25     logger.info("Main window is created");
26     init();
27   }
28
29   /**
30    * Creates the graphical interface
31    */
32   private void init() {
33     logger.info("Graphical user interface is initialized");
34     setTitle(PROGRAM_NAME);
35     setJMenuBar(createMenuBar());
36     add(createTableComponent(), BorderLayout.CENTER);
37     add(createFilterBar(), BorderLayout.NORTH);
38   }
39 }
```

The next two method calls are also owed to the base class *JFrame*. You use the universal method *add()* to add components to a window, in this case the main window. The first of the methods is passed the table area as the first parameter, and the orientation within the window as the second. The first parameter is again a method. It returns the table component and, like the *createMenuBar()* method, is only completed afterwards.

At the end of the *init()* method, add the filter bar to the main window (line 38). To do this, call the *add()* method again and pass the filter bar as the first parameter and the orientation as the second. Again, the first parameter is a method call that creates the filter bar. This method is also completed afterwards. What is missing now is the import statement in line 3 and the comments on the class and the methods. Complete everything as specified in the listing.

In figure 25.8 you can see the current working state of the main window with the alignment of the components. As with a directional map, the north is at the top of the window and the south is at the bottom. The Border Layout aligns all components to the border of another

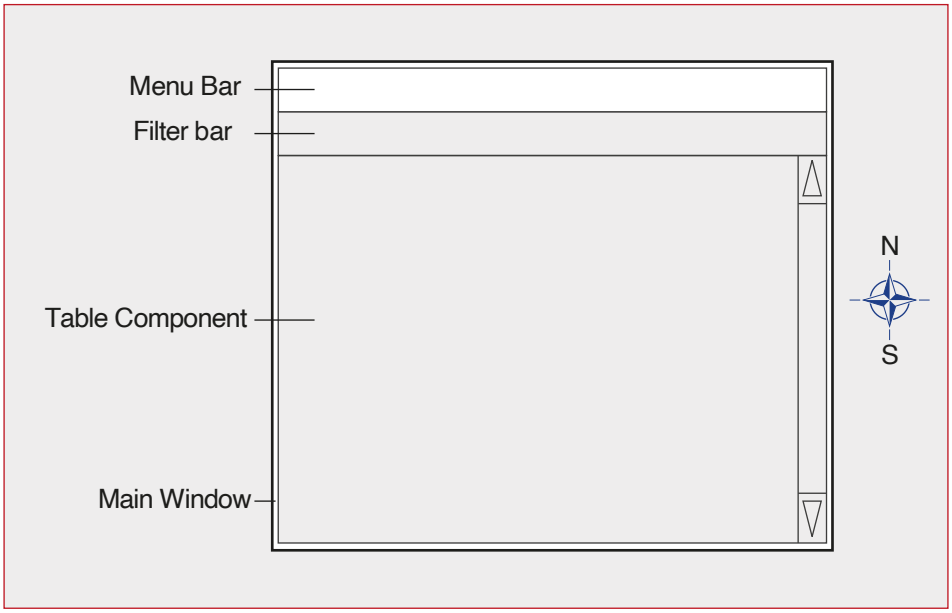


Figure 25.8: The border layout aligns components to the window border.

component, which in this case is the main window. The parameter *BorderLayout.NORTH* ensures that the filter bar will be located in the "north" of the window.

For the table component, the parameter *BorderLayout.CENTER* ensures that the table component will occupy the entire remaining interior space. Finally, the question remains why the menu bar will appear above the filter bar. This is because the method call *setJMenuBar()* always places it in the correct position at the top of the window, regardless of the layout, as is the case with almost all operating systems. No special layout is necessary for this.

To get the class *MainWindow* syntactically error-free, you need to create at least the wrappers of the three methods called in the method *init()*. You can leave the work to Eclipse again. To do this, place the mouse pointer over the missing methods again and wait until the editor displays a pop-up with the programming help. Select CREATE METHOD for each of the three missing methods in turn by clicking on the corresponding link of the pop-up.

Then save the entire project. It should now be syntactically error-free. However, you should not start it yet, because the implementation of the three methods is missing. Eclipse has only used *null* as return value in the method bodies. This would lead to a *nullPointerException* when starting the program. Instead of starting the program, we start by completing the *createMenuBar()* method.

Implementing Method »createMenuBar()«

The menu bar according to the blueprint at the beginning of the chapter shall consist of only one file menu. This should have two menu items. Proceed with the implementation of the menu bar again in the same way as with the method *init()* and assign to the individual steps during the construction of the program surface again own methods with speaking

names. This way you achieve that the program is structured cleanly and the source code is well understandable.

The method *createMenuBar()* can be seen in the listing 25.5. It starts again with a logging statement. This is followed by the creation of an object named *menueleiste* of Swing type *JMenuBar*. Also add the necessary import statement *import javax.swing.JMenuBar;* to the beginning of the file.

As with the window component of type *JFrame*, new components can be added to objects of type *JMenuBar* using the *add()* method. In this case the new component is the previously missing file menu. It is created by the *createFileMenu()* method on line 4 of the listing. Finally, replace the method's return statement generated by Eclipse by returning the finished object named *menubar* by return statement instead.

Listing 25.5: The method »createMenuBar()« creates the program's menu bar.

```
1  private JMenuBar createMenuBar() {
2    logger.info("Menu bar is created");
3    JMenuBar menuBar = new JMenuBar();
4    menuBar.add(createFileMenu());
5    return menuBar;
6  }
```

If you entered the Java code like this and then save it, Eclipse will call the *createFileMenu()* method as unknown. Hover over this method again, wait for the programming help to appear, and click CREATE METHOD 'CREATEFILEMENU()' to create a method body again.

Implement Method »createFileMenu()«

The implementation of this method is again nested, as you can see from the listing 25.6.

Listing 25.6: The method »createFileMenu()« creates the program's file menu.

```
1  private JMenu createFileMenu() {
2    logger.info("File menu is created");
3    JMenu fileMenu = new JMenu();
4    fileMenu.setText("File");
5    fileMenu.add(createFileOpenMenuCommand());
6    fileMenu.add(createExitMenuCommand());
7    return fileMenu;
8  }
```

The implementation starts again with a logging statement. This is followed by the creation of the file menu (line 3). For this, add the import statement *import javax.swing.JMenu;* at the beginning of the file again. In the next line, give the menu the name *setText()* via the *file* method. The next two statements create the two menu items via corresponding methods. At the end of the method, override the return statement again and return the file menu to the menu bar.

Now save the entire project. Eclipse should mark the two method *createFileOpenMenuCommand()* and *createExitMenuCommand()* as unknown. Place the mouse pointer on the first of the two methods and create a method body as before using the Eclipse programming guide.

Implement Method »createFileOpenMenuCommand()«

The implementation of the method can be seen in the listing 25.7. On line 2 of the method, create a new object named *fileOpenMenuCommand* of Swing type *JMenu Item*. Again, add an import command for this at the beginning of the file that imports the *javax.swing.JMenuItem* class. Then give the menu command for opening the file the name *Open...* . This is followed by the installation of an action listener via the *addActionListener*() method. Finally, for this class, add the import statement *java.awt.event.ActionListener* to the beginning of the file.

Listing 25.7: This method creates the »Open File...« menu command of the file menu.

```
1   private JMenuItem createFileOpenMenuCommand() {
2     JMenuItem fileOpenMenuCommand = new JMenuItem();
3     fileOpenMenuCommand.setText("Open File...");
4     fileOpenMenuCommand.addActionListener(new ActionListener() {
5       public void actionPerformed(ActionEvent e) {
6         //showFileOpenDialog();
7       }
8     });
9     return fileOpenMenuCommand;
```

An anonymous object is passed to the action listener as a parameter. To do this, we use an anonymous class in lines 5 to 8 that overrides the *actionPerformed()* method. The *actionPerformed()* method is triggered whenever the user triggers the menu command. Then a dialog should appear that allows the user to open a CSV file. In this respect, the method *showFileOpenDialog()*, which will be implemented later and is therefore still commented out in the listing, follows after this.

Now save the project again. Eclipse should show the call to the *createExitMenuCommand()* method as the only error at the moment. Let Eclipse create the body of this method using the programming help.

implement Method »createExitMenuCommand()«

Listing 25.8 shows the implementation of the class. It starts by creating the menu item on line 2, then giving the menu command the name *Exit*.

Listing 25.8: This method creates the "Exit..." menu item of the file menu.

```
1   private JMenuItem createExitMenuCommand() {
2     JMenuItem exitMenuCommand = new JMenuItem();
3     exitMenuCommand.setText("Quit");
4     exitMenuCommand.addActionListener(new ActionListener() {
5       public void actionPerformed(ActionEvent e) {
6         System.exit(0); // Exit 0 = normal program end
7       }
8     });
9     return exitMenuCommand;
```

The statement on line 4 again installs an action listener. Again, the anonymous class overrides the *actionPerformed()* method of the base class. When the menu command is triggered and the *actionPerformed()* method is called, the program executes the *System.exit(0)* statement. This statement corresponds to a normal program exit. Provided you have completed the implementation of this method, save your work now. Eclipse should be satisfied and no longer show any syntactical errors. However, in order for you to run the program without errors, it is necessary to implement the table component and filter bar.

Implementing Method »createTableComponent()«

Let's start with the implementation of the table component. The implementation of this method is shown in Listing 25.9. Start by correcting the type of the return value. Correct it to return an object of class *JScrollPane*. The second line of the listing shows the usual logging output of the program. This is followed by the creation of the table model, an object called *courseStatistics*. This variable is an attribute of the class and therefore must be created globally. After inserting the statement to do this, go to the variable highlighted by Eclipse and select CREATE FIELD 'COURSESTATISTICS' from the programming help pop-up. This causes Eclipse to create an attribute (object variable).

Listing 25.9: This method creates the table component.

```
1   private JScrollPane createTableComponent() {
2     logger.info("Table component is created");
3     courseStatisticsTableModel = new DefaultTableModel();
4     courseStatisticsTable = new JTable(courseStatisticsTableModel);
5     courseStatisticsTable.getTableHeader().setFont(new Font("SansSerif",
          Font.BOLD, 12));
6     return new JScrollPane(courseStatisticsTable);
7   }
```

In line 4 you can see how the actual table for the course statistics is created. In the constructor call, it gets passed the (empty) table model. In line 5, the program calls the *setFont()* method, having previously obtained the table headers via the *getTableHeader()* method. The *setFont()* method is used to set the font for the table column titles. Finally, on line 7, the program creates an anonymous object of type *JScrollPane* and returns this table component to the *init()* method.

Finally, complete the missing import statements for the *JScrollPane*, *DefaultTableModel*, *JTable* and *Font* classes. Also add the attributes for *coursestatistics* and *coursestatisticsTable*. To do this, also go to the highlighted errors in the Eclipse editor and wait until the programming help appears. For the imports, select IMPORT so that Eclipse displays the appropriate statement. For the missing attributes, select CREATE FIELD. For the table, unfortunately, you need to correct the type because Eclipse creates it incorrectly. The object must be of type *JTable*. After saving the project the class should be error free again.

Implementing Method »ceateFilterBar()«

Finally, you need to implement the *createFilterBar()* method. The source code for this can be found in Listing **??**. Again, start by correcting the type of the return value. Correct it so

that the method returns an object of class *JPanel*. As with the other methods, the logging output comes first, informing what the purpose of this method is. After that, the program creates the label for the filter panel. The filter field is created in line 4 as a simple text field. It will be replaced by a special field later. After that, the actual bar is created, to which the program adds the filter label and the filter field.

Listing 25.10: This method creates the table component.

```
1   private JPanel createFilterBar() {
2     logger.info("Filter bar is created");
3     JLabel filterDescription = new JLabel("Filter: ");
4     JTextField filterField =
5       TableFilter.createFilterField(courseStatisticsTable);
6     JPanel filterBar = new JPanel();
7     filterBar.add(filterDescription);
8     filterBar.add(filterField);
9     return filterBar;
```

When you have entered the source code according to listing 25.10 into the editor of Eclipse, you still have to add the missing import statements. To do this, place the mouse pointer again on the places that Eclipse marks as errors, wait until the programming help appears, and then select IMPORT. If everything is complete, save the project now and start the program again. If everything went well, a window should appear with a menu bar and a filter bar. The table is not displayed because the program has not read in any file yet and therefore the table model is still empty.

Implementing Method »showFileOpenDialog()«

To complete the class, return to the *createFileOpenMenuCommand()* method. In Eclipse, when you click this method in the OUTLINE window, Eclipse sets the editor cursor to the beginning of the method. In this method, remove the line comment before calling the *showFileOpenDialog()* method. Since this method is not implemented yet, Eclipse will mark it as an error afterwards. Put the mouse pointer back on the error and wait until Eclipse shows the programming help. It contains two options (Figure 25.9).

With the first option, Eclipse creates a method *showFileOpenDialog()* whose visibility is limited locally to the action listener. The second option, on the other hand, creates a method that is visible throughout the class. If you still want to call this dialog from somewhere else in the program, the first option is better. Now select this option and let Eclipse create a method body. After that, complete the implementation as shown in Listing 25.11.

The method source code starts with logging output that the dialog is called. Then the method creates an object of class *JFileChooser* - this is the dialog to open the file. Add the corresponding import statement. To do this, move the mouse cursor over the class and select IMPORT 'JFILECHOOSER' (JAVAX.SWING) from the programming help. In the line after that, the program calls the *setDialogTitle()* method to give the dialog a title.

To make sure that the user selects only those files that the program can handle, there is a swing class called *FileNameExtensionFilter*. In line 6, the method specifies CSV files for the filter. In line 7, the method passes an object of this class via the *setFileFilter()* method.

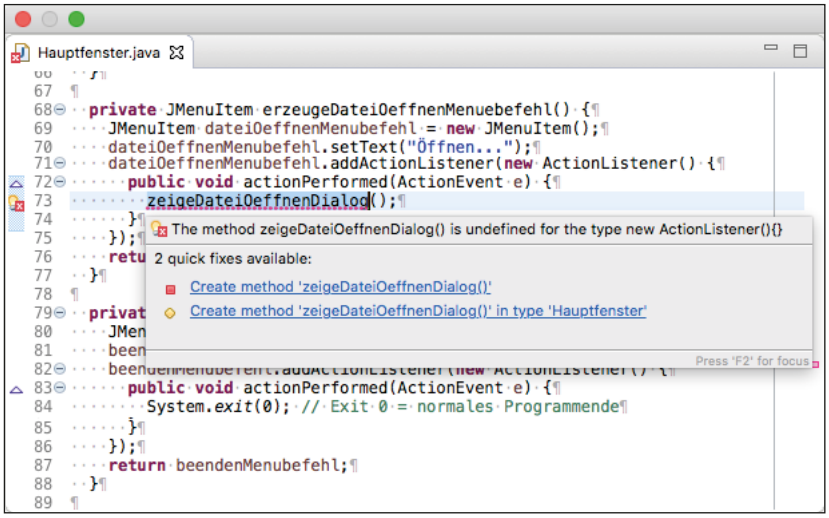


Figure 25.9: Eclipse provides two ways to implement the unknown method.

Again, complete the import statement for this class using Eclipse's programming guide as described earlier.

Listing 25.11: This method starts the dialog to open a file.

```
1  private void showFileOpenDialog() {
2    logger.info("File open dialog is called");
3    JFileChooser fileOpenDialog = new JFileChooser();
4    fileOpenDialog.setDialogTitle("Open Course Statistics");
5    FileNameExtensionFilter fileFilter =
6      new FileNameExtensionFilter("CSV Files", "csv");
7    fileOpenDialog.setFileFilter(fileFilter);
8    File csvDirectory = new File(System.getProperty("user.dir") +
9        System.getProperty("file.separator") + "csv");
10   fileOpenDialog.setCurrentDirectory(csvDirectory);
11   int returnVal = fileOpenDialog.showOpenDialog(MainWindow.this);
12   if (returnVal == JFileChooser.APPROVE_OPTION) {
13     File file = fileOpenDialog.getSelectedFile();
14     openCsvFile(file);
15   }
16 }
```

It annoys many users when they end up in a completely foreign directory when opening a file. For this reason, the method in line 8 determines the directory where the sample CSV files of this project are located. In the first step, the method determines the directory from which the program is started. Then, the method assembles the path for the directory within the project. Finally, in line 10, the method passes this directory to the dialog. It is important to add the corresponding import instruction *java.io.File* for the Swing class *File* with the programming help of Eclipse.

Via the method *showOpenDialog()* in line 11 the dialog appears on the screen. As parameter the method expects the parent of the dialog - this is the calling component. Where the dialog will appear on the screen depends on this parameter. For example, if you specify *null*, the program centers the dialog to the center of the screen, which is unusual. If you specify the main window, as shown in the listing, the program centers the dialog relative to the calling window of the program, which is correct.

On line 12, the method queries whether the dialog was closed with OK. If this is the case, it is clear that a file must be read in. Therefore, the method determines the file (line 13) and opens the CSV file using the *oeffneCsvFile()* method. Since this method does not exist yet, it is marked as an error.

Implementing Method »openCsvFile()«

Therefore, create a method body again using Eclipse's programming help. Then implement the method following the template of the listing 25.12. The first two lines of the listing compose the message for the logging output from the file path. This is followed by the actual logging output in line 4.

Line 5 then prepares the file model in the form of a vector. The vector data structure corresponds to the structure of a CSV file of *n* records with *n* data fields. The following try/catch block calls the *parseCsvFile()* method of the *CsvReader* class. The class is used to read the CSV file and output it as a vector. The vector currently still contains all records including the table header according to the analysis in section 25.3.2, »Reading the File«.

Listing 25.12: This method opens a CSV file.

```
1   private void openCsvFile(File file) {
2     String path = file.getAbsolutePath();
3     String loggerMessage = "Open CSV file: " + path;
4     logger.info(loggerMessage);
5     try {
6       CsvFileParser csvFileParser = new CsvFileParser();
7       Vector<Vector<String>> records =
8           csvFileParser.parseCsvFile(path);
9       Vector<String> columnHeaders = records.elementAt(0);
10      records.remove(0);
11      courseStatisticsTableModel.setDataVector(records, columnHeaders);
12      courseStatisticsTable.getColumnModel().getColumn(0).
            setPreferredWidth(30);
13      courseStatisticsTable.getColumnModel().getColumn(1).
            setPreferredWidth(20);
14      courseStatisticsTable.getColumnModel().getColumn(2).
            setPreferredWidth(20);
15      courseStatisticsTable.getColumnModel().getColumn(3).
            setPreferredWidth(200);
16      courseStatisticsTable.getColumnModel().getColumn(4).
            setPreferredWidth(30);
17      setTitle(PROGRAM_NAME + " - " +  file.getName());
18    } catch (FileNotFoundException e) {
19      logger.info(e.getLocalizedMessage());
```

```
20        String message =
21          "The file no longer exists. Please check\n" +
22          "the directory where the file is located\n" +
23          "and then try to open the file again";
24        showError(message);
25      } catch (IOException e) {
26        logger.info(e.getLocalizedMessage());
27        String message =
28          "The file is corrupted. Please check\n" +
29          "the file and try again afterwards\n" +
30          "to open the file.";
31        showError(message);
32      } catch (ArrayIndexOutOfBoundsException e) {
33        logger.info(e.getLocalizedMessage());
34        String message =
35          "The file is empty and therefore\n" +
36          "cannot be displayed.";
37        showError(message);
38      }
39    }
```

Next, the method creates a new object for the column headers (line 9) and then removes them from the vector. In this way, the method separates column headers from the actual values. In the next line, the method now sets the determined values for the column headers and the table contents as an object named *coursestatistics* (line 11). You may have noticed that almost all of the objects in this class were local variables. That's not the case here, because the object *kursstatistik* is a real attribute of the class, accessed from multiple places.

In the next line of the method, the method sets the name of the class as the name of the main window (line 12). This is followed by three catch blocks. The background is that several errors can occur when reading in a file. On the one hand, the file may have been deleted during the read-in, or it may be damaged or empty. All of these exception states are "caught" cleanly by the method here.

If you have implemented the method according to the listing template, now add the file imports of the classes *Vector*, *FileNotFoundException*, *IOException* and *ArrayIndexOutOfBoundsException* via Eclipse's programming help. Next, create the method body for the *showError()* method, also using Eclipse's programming help.

Implementing Method »showError()«

After that, start implementing the *showError()* method according to listing 25.13. The method calls a Swing window of class *JOptionPane* and outputs an error message. For this method to be error-free, you must add the import statement for the *JOptionPane* class.

Listing 25.13: This method displays error messages from the program.

```
1   private void showError(String message) {
2     JOptionPane.showConfirmDialog(this, message, "coursestatistics",
3         JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE);
4   }
```

After that, the project is still stupidly not error-free because the *CsvParser* class is missing. Put the mouse cursor on this class call and select CREATE CLASS 'CSVPARSER' from the programming help. In the dialog that appears afterwards, specify the package path *net.programmingcourse.coursestatistics.model* for the new class. Eclipse then creates another class, which you complete in the next section.

25.4.7 Implementing Class »CsvParser«

The *CsvParser* class has the task of reading in a CSV file and outputting it as a vector. Implement the method using the Eclipse editor, as shown in Listing 25.14. To parse CSV files, the class has a method *parseCsvFile()*. At the beginning of the method, it creates a stream object to read the file (line 19) and an object for the records (line 20).

The structure of the vector object exactly mirrors the structure of a CSV file: each record (row) is a vector of data fields (columns). The whole file is again a vector of records and thus results in a matrix (table structure). Why the class *vector*?

As mentioned in the 25.3.2 section, »Developing the Data Model«, vectors are ideal for this use case because they grow dynamically when a file is read. Thus, the number of rows and columns need not yet be known when the table is initialized. In addition, vectors have the advantage that a Swing table can be easily updated with them: It has a method that expects a vector as a parameter.

Listing 25.14: The class »CsvReader« reads a CSV file.

```
1  package net.programmierkurs.kursstatistik.model;
2
3  import java.io.BufferedReader;
4  import java.io.FileNotFoundException;
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.util.Arrays;
8  import java.util.Vector;
9  import java.util.logging.Logger;
10
11 public class CsvParser {
12
13   private static final Logger logger = Logger.getLogger("Course
         statistics");
14   private String separator = ";";
15
16   public Vector<Vector<String>> parseCsvFile(String csvFile)
17       throws FileNotFoundException, IOException {
18     logger.info("URL of CSV file:" + csvFile);
19     BufferedReader dateileser = null;
20     Vector<vector<string>> datensaetze = new Vector<vector<string>();
21     try {
22       String line = "";
23       dateileser = new BufferedReader(new FileReader(csvFile));
24       while ((line = dateileser.readLine()) != null) {
```

```
25          String[] datafields = line.split(separator);
26          Vector<String> dataset =.
27            new Vector<String>(Arrays.asList(datafields));
28          dataset.add(dataset);
29          logger.info("Dataset read and added.");
30        }
31      logger.info("Read dataset finished.");
32      } finally {
33        if (dateileser != null)
34          dateileser.close();
35      }
36      return dateileser;
37    }
38  }
```

After the data structures are built, a try/catch block follows. This is necessary because the file reader object, which is required to open the CSV file, can throw two exceptions: a *FileNotFoundException* and a *IOException*. The *FileNotFoundException* occurs in the (unlikely) case that someone deletes the file after it has been selected via the program's dialog. A *IOException* occurs when there are read errors, for example.

It is very important not to handle the exceptions in this class, but to pass them to the interface. If the class would handle these exceptions, they would not come to the surface, and the user would wonder why nothing is displayed in case of an error. You probably know this behavior from many applications. It is nothing but a sign of bad programming. You have to think carefully about what the user expects in such programs as course statistics and handle the exceptions accordingly.

Inside the try/catch block is a while loop that reads each line of the file until the end of the file is reached. Here, the method splits all lines of the file into individual data fields (line 24) and uses the separator, which is defined as a semicolon. Then the method assembles the data fields into a record vector (line 25/26) and inserts them into the entire vector object (line 27).

The try/catch block still has a finally branch. This is important to close the stream again after reading the file. But this is only done if the file could be opened at all and the object is not *null*. Finally, the method passes the records to the calling method (line 36). This completes the implementation of the class. The only thing missing now is the table filter, which the next section completes.

25.4.8 Implementing the Class »TableFilter«

To implement the class *TableFilter*, call the class *MainWindow* in the Eclipse editor. Then, in the OUTLINE window, select the *createFilterBar()* method, whereupon the editor places the cursor on this method. Rewrite lines 4 and 5 of this method to call a table filter, as shown in Listing 25.15.

Listing 25.15: The method »createFilterBar« with the call to the class »TableFilter«.

```
1   private JPanel createFilterBar() {
```

```
    logger.info("Filter bar is being created.");
    JLabel filter label = new JLabel("Filter: ");
    JTextField filterfield =.
      Tablefilter.createFilterField(coursestatisticstable);
    JPanel filterbar = new JPanel();
    filterbar.add(filter label);
    filterbar.add(filterfield);
    return filterpanel;
  }
```

After this action, Eclipse immediately marks the *table filter* class as faulty because it is missing from the project. Place the mouse cursor on this erroneous class call, wait for the programming help, and let Eclipse create a class body again in the package *net.programmingcourse.coursestatistics.ui*. Then overwrite the class with the implementation listed in Listing 25.16.

The very simple filter class consists entirely of the static method *createFilterField()*. The method is static because the filter is needed permanently during program execution. It would not make sense to create a new filter object each time the filter is run. The method is passed the table of the program. At the beginning it determines the table model, what should be sorted: the course statistics. If this is *null* and thus not initialized, the program terminates with an error, since a program with such a programming error should not start.

Listing 25.16: This class provides a very simple table filter.

```
package net.programmingcourse.coursestatistics.ui;

import java.util.logging.Logger;

import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.RowFilter;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;

public class TableFilter {

  private static final Logger logger = Logger.getLogger("Course
      Statistics");

  public static JTextField createFilterField(JTable table) {

    DefaultTableModel courseStatistics =
      (DefaultTableModel) table.getModel();
    if (courseStatistics == null) {
      logger.severe("Course statistics not initialized");
      throw new RuntimeException("Course statistics not initialized");
    }
    TableRowSorter<TableModel> sorter =.
```

```
27        new TableRowSorter<TableModel>();
28      sorter.setModel(coursestatistics);
29      table.setRowSorter(sorter);
30
31      final JTextField filterField = new JTextField(10);
32      filterField.getDocument().addDocumentListener((DocumentListener)
33          new DocumentListener() {
34        public void insertUpdate(DocumentEvent e) {
35          update(e);
36        }
37
38        public void removeUpdate(DocumentEvent e) {
39          update(e);
40        }
41
42        public void changedUpdate(DocumentEvent e) {
43          update(e);
44        }
45
46        private void update(DocumentEvent e) {
47          logger.info("Update the view");
48          String text = filterField.getText();
49          if (text.trim().length() == 0) {
50            sorter.setRowFilter(null);
51          } else {
52            sorter.setRowFilter(RowFilter.regexFilter("(?i)" + text));
53          }
54        }
55      });
56      return filterField;
57    }
58  }
```

If the table model exists, the method creates a new row sorter (line 26/27). Then the method passes the model to be sorted to the sorter and the table to the sorter (line 28/29). In the following line, the class creates an object called *filterfield*, which will later appear on the interface. For this object, the method adds a *DocumentListener*. As the name implies, this object listens for changes to the text field. It does this by implementing methods of the listener interface.

The first method *insertUpdate()* watches to see if the user has inserted anything into the text field (lines 34 to 36). The *removeUpdate()* method listens to see if anything has been removed (lines 38 to 40), while the *changedUpdate()* method watches for any changes to the field (lines 42 to 44). All methods delegate the response to the change of the text field to the *update()* method, which then completes the implementation of the class.

The *update()* method gets the text from the field, performs trimming to check if the user has entered unnecessary spaces. If not, the method sets the passed text as a filter of the table model *coursestatistics*. The *setRowFilter()* method has the effect that the program updates the table view immediately.

25.5 Test

To test the program, create a new folder within the project called *csv* by right-clicking on the project node *Swing Programs* and creating a new directory. Then right-click on the directory and select NEW → FILE from the context menu. Enter *CourseStatistics.csv* as the file name. Eclipse should then open the file with a system editor. Fill the file according to listing 25.17 and save it afterwards.

Listing 25.17: The test data for the CSV file.

```
Student Number; First Name; Last Name; Mandatory Subjects; Elective Subjects
100010; Anna; Seitz; Mathematics I, Programming I (Roth); English
100011; Julia; Lippert; Mathematics I, Programming I (Roth); Biology
100012; Lukas; Wittek; Mathematics I, Programming I (Roth); History
100013; Florian; Krause; Mathematics I, Programming I (Roth); Geography
```

Next, start the *CourseStatisticsApp* program and select FILE → OPEN from the program's menu. The dialog that appears after this should show the CSV directory you just created with the file *CourseStatistics.csv*. Select the CSV file and click OPEN. The program should then display the file in tabular form (Figure 25.10).

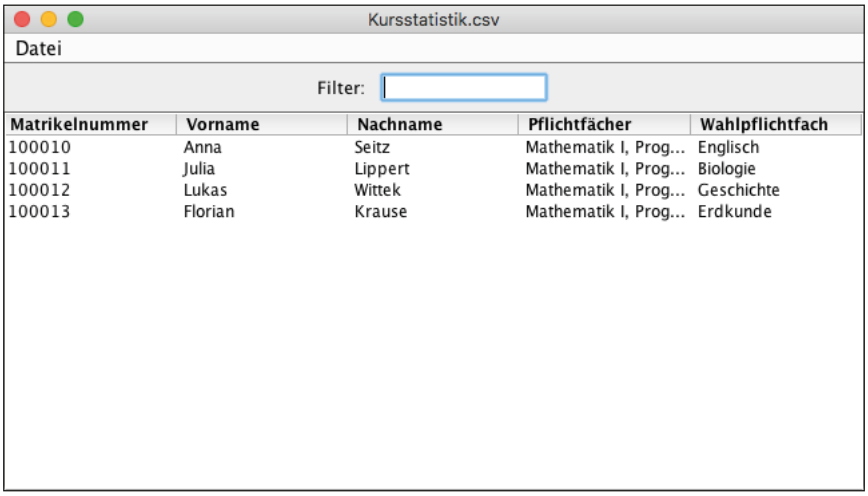


Figure 25.10: The complete application after loading the course statistics.

25.6 Deployment

To run the program without the Eclipse development environment, you must export it as a Java archive. To do this, right-click on the *Swing_Programs* project and select EXPORT from the context menu that appears afterwards.

In the dialog that appears, expand the JAVA branch and select export the program as RUNNABLE JAR (Figure 25.11). On the second page of the dialog, select the Launch Configuration of the project and enter as EXPORT DESTINATION *Swing_Programs/product/CourseStatisticsApp.jar*.

For larger projects with many libraries, it is important to export these libraries correctly so that the program can be run without a development environment. Eclipse offers three options when exporting:

- Extract required libraries into generated JAR
- Package required libraries into generated JAR
- Copy required libraries into a sub-folder next to the generated JAR

With the first option, Eclipse resolves the libraries and copies the classes individually into the new JAR archive. With the second option, Eclipse packages the libraries inside the new JAR archive, while with the third it places them separately in the export directory.

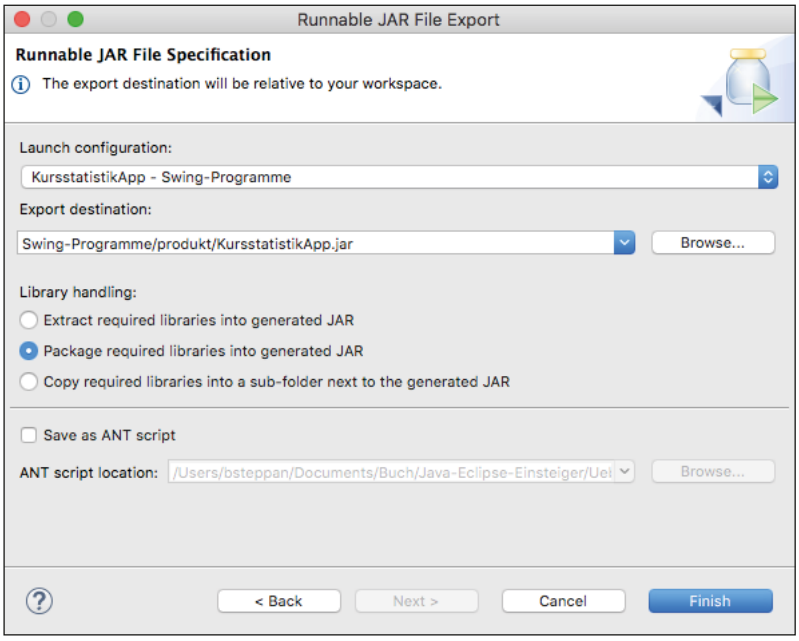


Figure 25.11: The export of the program as an executable file.

Since the *CourseStatisticsApp* doesn't need any libraries outside of the Java Runtime Environment, it doesn't matter which options you choose. In general, I would recommend option two for medium sized projects because a single JAR archive is easier to handle. For very large projects, option three is recommended because it keeps the Java archive small.

You can usually start the Java archive now by double-clicking on the executable file *CourseStatisticsApp.jar*. To set individual options, for example to allocate more main memory to the program, a start script is necessary. Listing 25.18 shows a simple start script for Windows. It assumes that you have set the search path to the virtual machine correctly.

Listing 25.18: The startup script to start the program on Windows.

```
@echo off
REM
REM Program: CourseStatisticsApp
REM Description: book "Getting started with Java using Eclipse"
REM Copyright: (c) 2022 by Bernhard Steppan
REM Author: Bernhard Steppan
REM Version 1.0
REM Please adapt the path to the JRE!
java -jar coursestatisticsApp.jar
@echo on
```

Enter the start script printed in Listing 25.18 in any text editor and save it under the name *CourseStatisticsApp.bat* in the project directory *product*. You can now start the program outside of Eclipse. This also completes the work on the *CourseStatisticsApp*.

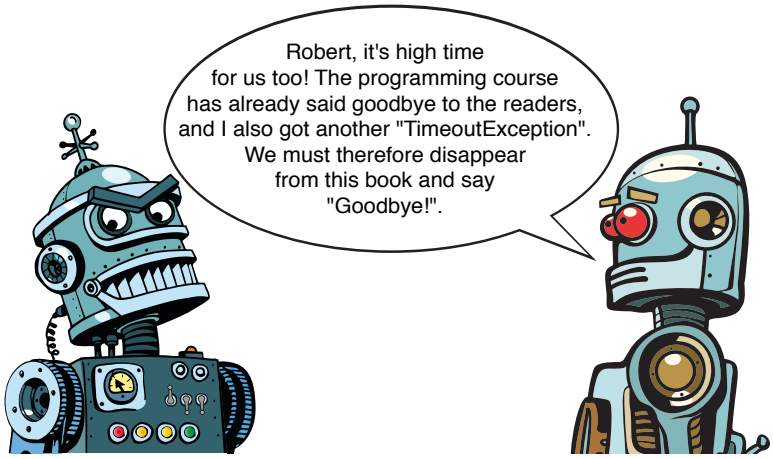


Figure 25.12: Robert and Roland are also saying goodbye now.

25.7 Summary

The course statistics in this chapter have given you an insight into programming graphical interfaces with Swing. You have seen how a simple interface with a menu bar and a table is created and how files are loaded, analyzed and displayed. This concludes our excursion into the world of Swing programming.

PART V

Appendix

26 Frequent Errors

26.1 Introduction

Java beginners are often frustrated because they fail because of errors whose cause they do not understand and which they cannot fix as a result. I have therefore compiled the most important error messages in this chapter and hope to provide some assistance in fixing these errors in your programs. The chapter starts with pure Java programming errors. This is followed by errors when using the Eclipse IDE.

26.2 Java Errors

26.2.1 Cannot Make a Static Reference To The Non-Static Field

Eclipse displays the error message *Cannot make a static reference to the non-static field* when you try to access an object variable from a static method (class method). An object variable may only be accessed from the object that has that attribute. So, the following code generates an error in line 11:

Listing 26.1: No object variable can be accessed from a static method.

```
1  //Examples/Frequent_Erros
2
3  package programmingcourse;
4
5  public class IncorrectAccessDemo {
6
7    int heightAnna = 171;
8
9    public static void main(String[] args) {
10     System.out.print("Anna is " + heightAnna + " cm tall");
11   }
12 }
```

To access the variable from the *main()* method, it must either be static (= class variable), or you create an object. The following program shows the access to a class variable on the method *main()*:

Listing 26.2: Access to a class variable.

```
1  //Examples/Frequent_Erros
2
3  package programmingcourse;
4
5  public class ClassVariableAccessDemo {
6
7    static int heightAnna = 171;
8
9    public static void main(String[] args) {
10     System.out.print("Anna is " + heightAnna + " cm tall");
11   }
12 }
```

If it does not make sense to redefine the object variable to a class variable for obvious reasons, the alternative is to create an object to access it. The following example shows this:

Listing 26.3: An instance variable can be accessed from an object.

```
1  // Examples/Frequent_Erros
2
3  package programmingcourse;
4
5  public class InstanceVariableAccessDemo {
6
7    int groesseAnna = 171;
8
9    public static void main(String[] args) {
10     System.out.print("Anna is " +
11       new InstanceVariableAccessDemo().groesseAnna + " cm tall");
12   }
13 }
```

In this example, the static method creates an object, then accesses the object variable *size* and outputs the following:

```
Anna is 171 cm tall.
```

26.2.2 Output of a »null« Value

If you forget to initialize a variable, the program outputs *null*. The following example calls a parameterless constructor to create an object. Later, it queries the name of an attribute of that object:

Listing 26.4: This program outputs the designation »null« instead of the name.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class IncorrectInitializationDemo {
6
7    public static void main(String[] args) {
8
9      Person julia = new Person();
10
11     System.out.print("The name of the person is " + julia.getName());
12
13   }
14 }
```

The program outputs the following:

```
The name of the person is null
```

Where is the error? The program looks completely harmless. The error is in line 7. The constructor is missing the parameter *name*. This can't be good. Because the program does not pass a value for the attribute *name* when it is created, this value remains *null*. The query in line 9 therefore also returns this value. This is what the correct program would look like:

Listing 26.5: This program provides the correct output of the name.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class CorrectInitializationDemo {
6
7    public static void main(String[] args) {
8
9      Person julia = new Person("Julia");
10
11     System.out.print("The name of the person is " + julia.getName());
12
13   }
14 }
```

26.2.3 NullPointerException

Even in professional Java programs *NullPointerExceptions* can occur. They always occur when an object was mistakenly not created. Eclipse usually warns about this. However, the development environment does not always succeed in detecting such errors in complex programs. The following program shows in an extremely simplified way how an NullPointerException occurs:

Listing 26.6: This program throws a NullPointerException.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class NullPointerExceptionDemo {
6
7    public static void main(String[] args) {
8
9      Person lukas = null;
10
11     System.out.print("The name of the person is " + lukas.getName());
12
13   }
14 }
```

The error occurs on line 9, where the creation of the object with the new operator is missing. Therefore, the program outputs the following:

```
Exception in thread "main" java.lang.NullPointerException
  at programmingcourse.NullPointerException.main(NullPointerException.java:9)
```

The output says that in line 9 of the program an attempt was made to access a variable to which the program has not yet assigned an object. If you remove the error in line 9, the program looks like this:

Listing 26.7: Creating an object in line 9 removes the NullPointerException.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class CorrectInitializationDemo {
6
7    public static void main(String[] args) {
8
9      Person lukas = new Person("Lukas");
10
11     System.out.print("The name of the person is " + lukas.getName());
12
13   }
14 }
```

The example seems banal. Who forgets to create such an obviously needed object? In larger programs, the error, as here in line 9, is unfortunately rarely so obvious. In professional programs, the creation of objects often depends on certain program conditions. If one has not thought through all these conditions completely, it happens that the program encounters a variable to which no object has been assigned yet. This variable then throws a NullPointerException as soon as the program accesses it. As of Java 14, by the way, the detection of such errors is made much easier because the program output in this Java version explicitly outputs the variable that is the cause of the NullPointerException.

26.2.4 Missing Break in Case Statement

Another very frequent error is caused by a forgotten break in a case statement. The following program converts one of the examples from chapter 14, »Conditional Statements«. A person's student state is to be used to infer whether he or she is a student or not:

Listing 26.8: Several breaks are missing in this program.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class MissingBreakDemo {
6
7    static String checkState(boolean student) {
8      int status = student ? 1 : 0;
9      String studentenstatus;
10     switch (status) {
11     case 0:
12       studentenstatus = "no student";
13     case 1:
14       studentenstatus = "student";
15     default:
16       studentenstatus = "unknown";
17     }
18     return studentenstatus;
19   }
20
21   public static void main(String[] arguments) {
22
23     // Person "Anna"
24     Person anna = new Person("State Anna: ", true);
25     String status = checkState(anna.isStudent());
26     System.out.println(anna.getName() + status);
27
28     // Person "Robert"
29     Person robert = new Person("State Robert: ", false);
30     status = checkState(robert.isStudent());
31     System.out.println(robert.getName() + status);
32   }
33 }
```

However, the program incorrectly outputs the following:

```
State Anna: unknown
State Robert: unknown
```

What causes the error? The error occurs because the break is missing in the case statements. Because it is missing, the program runs through each of the following case statements until it ends at the label *default*. In this label, the program deletes the *state*, resulting in the program output. A better way to do it is this:

Listing 26.9: This program stops in every case statement by the »break«.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class BreakDemo {
6
7    static String checkState(boolean student) {
8      int status = student ? 1 : 0;
9      String studentenstatus;
10     switch (status) {
11     case 0:
12       studentenstatus = "no student";
13       break;
14     case 1:
15       studentenstatus = "student";
16       break;
17     default:
18       studentenstatus = "unknown";
19       break;
20     }
21     return studentenstatus;
22   }
23
24   public static void main(String[] arguments) {
25
26     // Person "Anna"
27     Person anna = new Person("State Anna: ", true);
28     String status = checkState(anna.isStudent());
29     System.out.println(anna.getName() + status);
30
31     // Person "Robert"
32     Person robert = new Person("State Robert: ", false);
33     status = checkState(robert.isStudent());
34     System.out.println(robert.getName() + status);
35   }
36 }
```

This program now provides the correct output:

```
State Anna: student
State Robert: no student
```

26.2.5 Incorrect Comparison

Another very common error is to confuse the comparison operator with the assignment operator. In many cases this is already detected by Eclipse, as the following example shows:

Listing 26.10: Eclipse does not compile this program.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class IncorrectIntComparisonDemo {
6
7    public static void main(String[] arguments) {
8
9      int heightAnna = 171;
10
11     int heightJulia = 172;
12
13     if (heightJulia = heightAnna)
14       System.out.println("Julia and Anna are the same height");
15     else
16       System.out.println("Julia and Anna are not the same height");
17   }
18 }
```

Eclipse does not translate the program because instead of using a *boolean* type comparison, the program uses an assignment that results in an int value in line 13. The Eclipse error message is therefore also *Type mismatch: cannot convert from int to boolean.* The situation is different when using the comparison operator:

Listing 26.11: This program uses the correct comparison.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class CorrectIntComparisonDemo {
6
7    public static void main(String[] arguments) {
8
9      int heightAnna = 171;
10
11     int heightJulia = 172;
12
13     if (heightJulia == heightAnna)
14       System.out.println("Julia and Anna are the same height");
15     else
16       System.out.println("Julia and Anna are not the same height");
17   }
18 }
```

This program now provides the correct output:

```
Julia and Anna are not the same height.
```

Imagine, however, that the program is to perform a comparison with truth values. This would look like the following, for example:

Listing 26.12: Although this program is buggy, it does translate Eclipse.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class IncorrectBooleanComparisonDemo {
6
7    public static void main(String[] arguments) {
8
9      boolean stateAnna = true;
10
11     boolean stateRobert = false;
12
13     if (stateAnna = true)
14       System.out.println("State Anna: student");
15     else
16       System.out.println("State Anna: No student");
17
18     if (stateRobert = true)
19       System.out.println("State Robert: student");
20     else
21       System.out.println("State Robert: no student");
22
23   }
24 }
```

This program is compiled by Eclipse without any problems, but gives the following erroneous output:

```
State Anna: student
State Robert: student
```

What is the problem this time, since Eclipse does not complain? In lines 13 and 18 the program does not perform a comparison, but assigns new values to the variables *stateAnna* and *stateRobert*. This is allowed, but completely nonsensical. This is the reason for the erroneous output. The program looks better after a small change in line 13 and 18:

Listing 26.13: This program correctly queries the status of students.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class CorrectBooleanComparisonDemo {
6
7    public static void main(String[] arguments) {
8
9      boolean stateAnna = true;
10
11     boolean stateRobert = false;
12
13     if (stateAnna == true)
```

```
14        System.out.println("State Anna: student");
15      else
16        System.out.println("State Anna: No student");
17
18      if (stateRobert == true)
19        System.out.println("State Robert: student");
20      else
21        System.out.println("State Robert: no student");
22
23    }
24  }
```

Since the program uses the relational operator this time, the program output is also correct:

```
State Anna: student
State Robert: no student
```

This error is not as rare as many people believe. But it is relatively difficult to find and makes for completely chaotic program states.

26.2.6 Unhandled Exceptions

Even in professional programs one sees unhandled exceptions again and again. An example for this:

Listing 26.14: Dieses Programm fragt den Status der Studenten korrekt ab.

```
1  //Examples/Frequent_Errors
2
3  package programmingcourse;
4
5  public class UnhandledExceptionDemo {
6
7    public static void main(String[] args) {
8
9      ProgramLogic logic = new ProgramLogic();
10
11     try {
12       for (int state = 0; state < 4; state++) {
13         System.out.println("Check state: " + state);
14         logic.checkState(state);
15       }
16     } catch (IncorrectStateException fehlerobjekt) {
17
18     } catch (UnknownStateException fehlerobjekt) {
19
20     }
21   }
22 }
```

In the example, no handling can be seen in lines 17 and 19. However, exceptions are exceptions that must be handled in programs. To omit the handling is not a solution, but negligent.

26.2.7 NoClassDefFoundError

A *java.lang.NoClassDefFoundError* is raised when a class that a Java program wants to access with a new operator does not exist. This never occurs as long as you develop the program with Eclipse. Eclipse checks to make sure that all referenced classes exist. If you export the program so that it can be run without Eclipse, such an error may occur. It always occurs when the Java runtime environment did not find a certain class. In such a case, check the export of your program. For example, did you forget a library during the export?

26.2.8 ClassNotFoundException

Very related to the previous error is the *java.lang.ClassNotFoundException*. This error also occurs when the Java runtime environment cannot find a particular class. It is thrown when the program does not load a class with the new operator but, for example, with the *Class.forName(org.postgresql.Driver)* statement, where *org.postgresql.Driver* is the desired class to be loaded. In this case, the development environment cannot check whether the class exists in the project. In case of such an error, check if the search path to the class is set incorrectly or if an error happened during export.

26.3 Eclipse Errors

26.3.1 Eclipse Could Not Be Started

Cannot start Eclipse and the following error message appears *A Java Runtime Environment (JRE) or Java Development Kit (JDK) must be available in order to run Eclipse. No Java virtual machine was found after searching the following locations: ...* means that Eclipse did not find a virtual machine. This can happen for example if there is no Java installed on the machine.

Solution: Check if you have Java installed. If you do, check the search path to your installed Java version. As an emergency solution, you can also copy the complete JDK into the Eclipse"=folder. Eclipse will then find the correct Java version in any case.

26.3.2 Chaotic Eclipse Perspective

If you have broken Eclipse and can no longer find certain windows, it definitely helps to restore the perspective to its original state. To do this, use the PERSPECTIVE → RESET PER-

SPECTIVE command of the Eclipse WINDOW menu. In the dialog that follows this command, click the RESET PERSPECTIVE button to reorder the perspective.

26.3.3 Missing Window

Sometimes you have carelessly closed a window in the development environment, but you want to bring it back on the screen. This works with the command WINDOW → SHOW VIEW. If the window is not under the menu, you can use the command WINDOW → SHOW VIEW → OTHER... to bring up a dialog that shows all the Eclipse windows and from which you select the window that should appear.

26.4 Summary

The chapter has introduced you to some of the most typical Java and Eclipse errors. You can find more tips about Java and Eclipse on my homepage *https://www.programmingcourse.net*.

26.5 Literature

Paul Javin: Difference between ClassNotFoundException vs NoClassDefFoundError in Java; *https://javarevisited.blogspot.com/2011/07/classnotfoundexception-vs.html*

27 Glossary

Abstract class

No objects can be created from an *abstract class*. It is intentionally incomplete and forms the basis for further concrete subclasses.

Abstract method

An incomplete method for which only a signature is defined, but no statement sequence. An abstract method lacks implementation.

Class

The class is a template, a template for objects.

Class attribute

Class attribute is a synonym for *class variable*.

Class method

A class method is immutable (static) for all *examples* of a class. Class methods are declared by the keyword *static*.

Class operation

Synonym for *class method*.

Class variable

A class variable is immutable for all *examples* of a class. Class variables are declared by the *static* keyword.

Concrete class

A class from which objects can be derived.

Generalization

The term generalization is synonymous with *inheritance*.

Instance

Instance usually means an object. In computer science, a program that has been started is often called an instance.

Instantiation

The creation of an *object*, that is, an *instance* from a class.

Object

A special user-defined variable that encapsulates data and methods.

Object variable

Variable that is valid only for a special object.

static

The Java keyword *static* is used to indicate a *class variable* or a *class method*.

super

You use the Java keyword *super* to call the constructor of the base class (super class). It is a special class variable that the compiler automatically creates when it compiles a class.

this

The Java keyword *this* is used when you want to call the object itself. It is a special class variable that always points to the object. The compiler creates it automatically when it compiles a class.

Index

A

abstract 90
Abstract class 157, 485
Abstract Method 485
Abstract Windowing Toolkit 373
Abstraction 30
Access levels 398
Access Protection 34, 194, 398
Accuracy 126
Activities 310
Addition operator 216
Aggregation 37
Algorithms 427
– use 438
Analysis 310
And-operator 226
annotation 150
Anonymous class 150
Applets 12, 386
Arithmetic operators 214
Arrays 181
Assembler language 4
assert 90
Assignment 113
Assignment operator 228
Association 37
Attribute 26
Attributes 26
AWT 373

B

Base Class 32
Bitwise operators 228
Block 116
Block Comments 282
boolean 90, 134
Border Layout 376
break 90
Build System 68
byte 90, 128

C

Cannot make a static reference ... 473
case 90
Cast operator 231, 413
catch 90
char 90, 134
Character Data Type 134
Characters 134
Class 23, 26, 163, 256, 485
class 90
Class attribute 485
Class Import 256
Class Libraries 353
Class method 485
Class methods 191
Class operation 485
Class variable 485
Class Variables 102
ClassNotFoundException **482**
Comparative operators 222
Compile 321
Compiler 7, 68, 321
Composition 38
Concrete Class 145
concrete class 485
Conditional statements 237
Configuration Files 370
const 90
Constant 27
Constants 104
Construction Phase 308
Constructor 28
Constructors 198
Constructors without parameters 199
Container 372

continue 90
Coprocessor 126
Create objects 146

D
Debugger 68
Declaration 111, 146
default 90, 399
Default constructors 198
derive 152
Derived classes 33
Design 310
Design Flaws 40
Design Rules 43
Destructor 28
Destructors 203
Developing Algorithms 428
Development Environment 7, **45**
Development Processes 307
Difference operator 217
Division Operator 218
do 90
Do Loop 250
Documentation Comments 283
double 90, 133
Dynamic import 256
Dynamic Polymorphism 42

E
Eclipse Development Environment 50
Eclipse IDE **50**, 50
Eclipse Plug-ins 69
Eclipse Project Management 62
Editor 7, 63
else 90
Encapsulation 34, 194, 398
Enterprise JavaBeans 389, 390
Entity Beans 390
enum 90
Enumeration Type 170
Equality Operator 223
Event Control 386
Event Handling 373
exports 90
extends 90, 152

F
false 90
FileReader 368
FileWriter 369
final 90, 152, 156
finalize 203
finally 90
Fixed Point Number 128
float 90, 132
for 90
For Loop, extended 252
For Loop, simple 251
Function 204

G
Garbage collector 343
Generalization 38, 486
Generic class 163
Generics 163
Getter Methods 29, 205
Glossary 485
goto 90
Greater operator 225
Greater than or equal to operator 225
GridBag Layout 379

H
Help System 73
High-level programming language 5
Home Interface 390
HotJava 12
HotSwap 68

I
Identifier 27
if 90
If Statement 238
implements 90
Import 256
import 90
Import statement 256
Inequality Operato 223
Inheritance 31, 486
inheritance 152
Inner class 147
Installation Eclipse IDE **50**
Instance 25, 146, 486
instanceof 90
Instances 146
Instantiation 486
int 90, 130
Integer 128
Integer Literals 128
Interface 159
interface 90
Interfaces 159

J
Java 2 Micro Edition 390

Java Class Libraries 353
Java Database Connectivity 386
Java Editor 63
Java Enterprise Edition 389
Java Keywords 90
Java ME 390
Java Micro Edition 390
Java Native Interface 387
Java Runtime Environment 335
Java Standard Edition 355
java.base 356
JavaBean 386
JDBC 386
Jigsaw 13
JNI 387
JRE 335
JSE 355
JVM 340
JVM Configuration 349

K
Keyword 90
Keywords 90

L
Lambda operator 234
Layout Managers 376
Less operator 224
Less or equal operator 224
Line comments 282
Local Class 149
Logical Operators 226
long 90, 131
Loops 247

M
Machine language 4, 5
Message Driven Beans 390
Method 485
method 486
Method body 195
Method Call 120
Method implementation 195
Method Overloading 418
Method Overriding 420
Methods 28, 189
Model 41
Modeling 41
Modular Structure 69
Module 260
module 90
Modules 260
Modulo operator 219
Multiple Inheritance 33

N
Namespace 258
Namespaces 258
native 90
New 146
new 90
New Operator 230
NoClassDefFoundError **482**
Non-operator 226
NPE 475
null 90
NullPointerException 475

O
Oak 12
Object 23, 25, 357, 486
Object variable 99, 486
Object Variables 99
OOA/OOD 41
OpenJDK 13
Operation 29, 204
Operation Phase 308
Operator 213
Or-operator 227
Overloading 418
Overriding 420

P
Package 256
package 90
Package Explorer 62
Packages 256
Parameter 98
Persistence 41
Perspective 61
Perspektive 61
Planning Phase 308
Polymorphism 41, 418
Postdecrement operator 221
Postincrement operator 220
Predecrement operator 221
Preincrement operator 219
Prevent Overriding 423
private 90, 399
Product operator 218
Programming 3
Project management 7
Properties 26, 370
Property 26

protected 90, 399
public 90, 399

Q
Question mark operator 229
Quotient operator 218

R
Refactoring 41
Relationship 36
Relationships without inheritance 36
Release memory 343
Remote Interface 390
Remote Method Invocation 388
Requirements 310
requires 90
Restructuring 41
return 90
RMI 388
Runtime 366
Runtime Environment 335
Runtime environment 8

S
Session Beans 390
Setter Methods 28, 208
short 90, 129
Sign 126
Software Update 72
Sorting 429, 438
State 27
Stateful Session Beans 390
Stateless Session Beans 390
Statements 109
States 27
static 90, 191, 486
Static Polymorphism 42
strictfp 90
String 359
StringBuffer 363
Sun Microsystems 12
super 90, 155, 486
Superclass Object 357
Swing 373, 382
Swing Programs 445
switch 90
Switch statement 241
synchronized 90
System 364

T
Test 310
this 90, 100, 154, 486
threads 367
throw 90
throws 90
Tools 311
transient 90
Translate 321
true 90
try 90
Type Casting 413
Type violation 163
Types of Algorithms 429
Types of loops 248

V
Value Assignment 115
Value range 126
var 90
Variable 27
Variable Call 119
Virtual machine 340
void 90
volatile 90

W
while 90
While loop 249
Workbench 60
Workspace **70**
Wrapper Class 363
Wrapper Classes 361
Write Files 368

Getting Started With Java Using Eclipse

Mastering the Language and the Development Platform

Many people would like to learn Java but getting started is not easy since programming with Java requires at least two things: mastering the programming language and the development environment. With the help of many examples, this book shows how the language is structured. In addition, it employs the Eclipse development environment as an example of a powerful tool to teach developing Java programs.

In Basics, the first part of the book, you acquire your Java and Eclipse basic knowledge. This part lays the programming foundations, gives you an overview of Java technology, and shows you what is special about object-oriented programming.

In the second part called Java Language, everything revolves around the subtleties of the Java language and this is where the first small Java applications are created, aided by a fine blend of the knowledge part and practical exercises.

Java Technology is both the name and the focus of the third part which also introduces you to the rules to observe when programming, what class libraries are and what advantages they have. In addition, you will learn how to test programs, what algorithms are, and how to program them.

The fourth part, Java Projects, enables you to apply all the previous elements in an application with a graphical user interface. The project shows how to develop a larger application piece by piece with the Eclipse development environment. The Appendix concludes with a section on frequent errors that can occur when working with Eclipse, and a Glossary.

All sample programs discussed in the book, and bonus material, are contained in an archive file which is available free of charge from the Publishers' website www.elektor.com/books. On this site, search for: Java Eclipse. The tasks and solutions mentioned in the book are at: www.programmingcourse.net/courses/java_with_eclipse/.

Bernhard Steppan is a Java developer from the very beginning. He has written several C++ and Java books as well as numerous articles on programming languages, IT architecture, development tools, and other IT topics. Bernhard Steppan lives in Wiesbaden (Germany) and works as Chief IT Architect for DB Systel, the systems house of Deutsche Bahn.

Elektor International Media BV
www.elektor.com

ISBN 978-3-89576-561-2
9 783895 765612